

CLASSIFICATION OF CIFAR-10 IMAGES VIA CONVOLUTIONAL NEURAL NETWORK AND
DEEP NEURAL NETWORK MODELS

Steve Desilets

MSDS 458: Artificial Intelligence and Deep Learning

October 22, 2023

1. Abstract

Computer vision algorithms capable of identifying and classifying objects have revolutionized the automotive industry and empowered innovative companies to usher in the era of autonomous vehicles. In this study, we leverage the Canadian Institute for Advanced Research dataset (CIFAR-10) to create ten such computer vision models capable of classifying images into ten classes of objects that cars would commonly encounter while driving. Throughout the construction of these models, we examine the important impacts that model architecture and regularization have on model performance and we examine the features extracted by these models via analyses of their hidden nodes' activated values. Like previous researchers who paved the way for this study at tech and automotive companies, we find that convolutional neural networks with proper regularization can serve as excellent tools for classifying images into their corresponding classes.

2. Introduction

Currently, there are over 30 million autonomous vehicles on the road worldwide thanks to innovative tech companies and automakers like Waymo, Honda, Mercedes-Benz, and Deeproute ai (Elton 2023; Wikipedia 2023). One of the key breakthroughs that have enabled this accelerating commercial success of autonomous vehicles is the increasing sophistication of computer vision algorithms capable of identifying important features in a car's path. In this study, we will construct and examine the effectiveness of 10 computer vision algorithms similar to those leveraged by automakers to identify objects so that their driving algorithms can maximize passenger safety and vehicle efficiency. Instead of using video data captured by vehicles, though, we will leverage the CIFAR-10 dataset, which consists of 60,000 32 pixel by 32 pixel images. These images each correspond to one of ten types of objects that an autonomous vehicle would commonly encounter and need to identify in order to react appropriately.

Table 1 summarizes these ten image classes.

Table 1: Image Classes and Corresponding Labels in the CIFAR-10 Dataset

Class Label in CIFAR-10 Dataset	Class Description
0	Airplane
1	Automobile
2	Bird
3	Cat
4	Deer
5	Dog
6	Frog
7	Horse
8	Ship
9	Truck

Leveraging this data, we construct ten deep neural network and convolutional neural network models via varying architecture and regularization methodologies so that we can gain a better understanding of the impacts of model design on performance and so we can delve into the features extracted by the nodes in the model hidden layers and examine their efficacy at discriminating between image classes.

3. Literature Review

Computer vision algorithms have existed for many decades and have steadily increased in popularity as their accuracy rates have soared thanks to academic breakthroughs (Štević 2022). Some of the brilliant researchers who have catapulted the field forward include Dr. Lawrence Roberts, who proposed a method for extracting information about 3D objects from 2D images in 1963, and Dr. Kunihiko Fukushima, who, in 1979, developed the neocognitron – which recognizes patterns using convolutional layers in networks. More recently, in 2012, AlexNet dramatically improved upon the accuracy of prior computer vision classification problems at the ImageNet Competition, ushering in an exciting era of computer vision advancements, including state-of-the-art technology for facial recognition, medical diagnostics, and autonomous vehicles. In this study, we aim to address an image classification problem that is highly relevant to the field of autonomous vehicles using images from the CIFAR-10 dataset, which contains 60,000 images that each correspond to one of ten classes (Krizhevsky 2017). While existing, pre-trained models that focus on classification of this dataset’s images reach as high as nearly 99%, in this paper, we aim to determine whether we can exceed these impressive levels of

performance using deep neural networks or convolutional neural networks (tanlq 2022). Furthermore, we strive to build upon the work of prior researchers by taking a closer look at the drivers of model performance including model architecture, hyperparameter values, and activated layers.

4. Methods

For this study, we constructed 10 neural networks throughout the two primary phases of experimentation with CIFAR-10 image classification models. To begin this process, we first imported the CIFAR-10 data and conducted exploratory data analysis and data cleaning exercises, like examining the shape of the data and visualizing a subset of the images. We also preprocessed the data by dividing the 60,000-image dataset into a 45,000-image training dataset, a 5,000-image validation dataset, and a 10,000-image testing dataset, and by rescaling the RGB pixel values, which ranged from 0 to 255, to be between 0 and 1.

For the first phase of experimentation, we wanted to focus on identifying the relative performance strengths and weaknesses of deep neural networks (DNNs) and convolutional neural networks (CNNs) with varying architectures. Accordingly, in this round of experimentation, we constructed two DNN models and two CNN models. The first and second DNN models contained two and three hidden layers, respectively, while the third and fourth models (which were CNNs) contained two and three pairs of convolutional and max pooling layers, respectively. The final layer of each of these models was a softmax layer capable of classifying observations into one of the ten classes. The intent of this exercise was to determine the impacts of model type and layer architecture on model performance, so that we could work on improving the best of these four models during the second phase of experimentation. Notably, for each of these four models, we utilized no regularization techniques. After constructing these models, we evaluated the model's performance on the testing dataset and examined the features extracted by hidden layers by visualizing their activated values using ten testing images (one image from each of our ten classes).

The second phase of experimentation focused upon building upon the success of the first experimentation phase by working to improve the best model from the first phase – the CNN with three

pairs of convolutional and max pooling layers. Accordingly, in each of the subsequent six experiments, we attempted to improve the model by either adding complexity or by introducing regularization – depending upon the results of the previous model. For the fifth model, we introduced regularization by implementing early stopping with a patience level of three steps and by simplifying the model architecture by reducing the number of nodes in the flattened layer just before the final softmax layer. For the sixth model, we aimed to improve upon the fifth CNN model by modifying it to also leverage batch normalization with batch sizes of 500 images. Subsequently, for the seventh model, we built upon the success of the sixth model by adjusting it to perform L2 regularization with a regularization parameter of 0.01. For the eighth model, we abandoned the L2 regularization and instead aimed to improve the sixth model by introducing four dropout layers (after the max pooling and flattened layers) that each had dropout rates of 30%. For the ninth CNN model, we strived to improve the eighth model by implementing L1 regularization – this time leveraging a much lower regularization parameter of 0.01. For the tenth and final CNN model, we removed the L1 regularization and instead aimed to regularize the eighth CNN model by reducing the number of nodes in the final, flattened layer of the model. For each of these six CNN models, we evaluated model performance metrics (such as precision, recall, F1 score, and accuracy), examined performance summaries (like confusion matrices and accuracy / loss plotted by epoch during training), and analyzed the extracted features by visualizing activated values from the hidden max pooling layers.

5. Results

After creating our first model – the DNN with two hidden layers – for our first phase of experimentation, the primary takeaway from the results is that this model didn't perform particularly well. While the testing accuracy of 45.1% (as displayed in Appendix A) certainly exceeded the 10% accuracy that would have resulted from randomly guessing answers, the performance is certainly not great. Furthermore, as displayed in Appendix C, examination of the charts that plot accuracy and loss across epochs during model fitting reveals a major need for regularization that appears after just a few epochs since we leveraged no regularization techniques. In fact, the training accuracy remained about 10

percentage points higher than the validation accuracy throughout all of training. Notably, the precision and recall also vary widely by image class with the precision for trucks, 0.66, being over double the precision for birds, 0.29, and with the recall for ships, 0.72, being over double the recall for cats, 0.32.

The results from the second DNN model – the DNN model that leveraged three hidden layers and no regularization techniques – unfortunately conveyed worse performance than the results from the first model. As displayed in Appendix A, the testing accuracy fell by two percentage points to 43.1%. To make matters worse, as displayed in Appendix B, the most common misclassification resulting from this model was between automobiles and trucks – with 459 such misclassifications arising. Within the context of autonomous vehicles, it's very important for our model to be able to distinguish between these classes since these vehicles perform differently on the road. Like the first model, this model would clearly benefit from regularization as displayed by the accuracy and loss trendlines across epochs during model fitting, which exhibit gaps between the training and validation trendlines. Notably, there also is a massive range in some of the performance metrics – most notably the recall of 0.18 for deer being roughly one fourth the recall for trucks of 0.73. Overall, this model was probably the worst of the ten models constructed and underscores the point that sometimes adding model complexity does not actually improve model results.

As reflected in Appendix A, the third model – the CNN with two pairs of convolutional and max pooling layers with no regularization - exhibited a significant jump in performance compared to the first two DNN models. Specifically, the testing accuracy jumped up by 11 percentage points to 54.5% in this model. Despite this success, there was definitely still room for improvement. For example, the confusion matrix in appendix B conveyed that the model still struggled to distinguish between cars and trucks – making 660 such misclassifications and roughly half of the activation layer visualizations for the various channels remained inactivated, which suggests that the channels were not finding useful patterns for classifying images in our sampled images particularly often. Furthermore, the plot of training and validation accuracies across epochs during fitting suggest that this model desperately needs regularization – with training accuracy reaching 0.30 higher than validation accuracy by the end of model fitting.

Our fourth model and final model from phase one of experimentation – the CNN with three pairs of convolutional and max pooling layers with no regularization – was our best performing model from phase one. The testing accuracy reached 58.5%, which further underscores the point that CNNs can outperform DNNs at computer vision tasks since they can more aptly take advantage of spatial correlations between pixels. This model also seemed to make important improvements in our efforts to distinguish between cars and trucks – with roughly half the number of such misclassifications occurring in this model compared to the previous model for the testing dataset. Still, there seem to be differences in how well this model identifies certain classes. Specifically, the layer activation visualizations for the various channels seem to pick up on patterns from birds, planes, and horses more clearly than they do for other image classes. Also, there exist big gaps in performance metrics across classes such as the F1 score of 0.26 for cats and the F1 score of 0.73 for automobiles. Like the previous models, this model also clearly requires regularization since training accuracy reached 16 percentage points higher than validation accuracy by the end of training.

Since model four was the best performing model from the first phase of experimentation, all subsequent models constructed for phase two were CNNs with three pairs of convolutional and max pooling layers, and we aimed to iteratively improve these models via regularization. For model five, we attempted to improve the model by introducing early stopping regularization and by reducing the number of nodes in the flattened layer from 384 to 256. These regularization methods allowed the testing accuracy to jump roughly 5 percentage points up to 63.1%. This increase in accuracy was further highlighted by the continuation of the trend of fewer channels of the layer activation visualizations being inactivated as we advance the sophistication of our models. Still, this model definitely required more regularization given that its training accuracy was 12 percentage points higher than its validation accuracy by the end of training. Furthermore, we see that this model definitely struggles to distinguish between cats and dogs – making 563 such misclassifications on the testing dataset – which may be important for autonomous cars to distinguish since one animal may be more likely to dart into the road unexpectedly than the other.

For the sixth model, we aimed to improve upon the success of the fifth model by altering it to also leverage batch normalization with batch sizes of 500, and the results were awesome. This model produced the highest training accuracy of all models developed: 82.5%. Though the testing accuracy of 69.4% was lower than that, this was still the highest testing accuracy we'd achieved in the first six models, which suggests that batch normalization was definitely helpful for our regularization efforts. The activation layer visualizations also dramatically improved with this model to the point where nearly all of them were activated for all ten images that we inspected. The model also produced the best-looking confusion matrix of the first six models, and importantly, the precision for automobiles climbed to 85%. Still, the need for regularization, as evidenced by the gap between training and validation accuracy throughout model fitting, remained strong with this model.

Accordingly, for model seven, we aimed to build upon the success of the sixth model by modifying it to include L2 regularization, but unfortunately, the performance of this model actually declined with the testing accuracy falling to 66%. Some of the image classes exhibited particularly weak performance metrics, such as the testing F1 score for cats of 0.41 (which, for comparison, was much lower than the testing F1 score for ships of 0.78). In addition, the need for regularization still persisted with in this model as evidenced by the validation accuracy of 66% being 11 percentage points lower than the training accuracy of 77% at the end of model fitting. These findings emphasize the idea that L2 regularization seemed to not really benefit our model fitting process, so we removed L2 regularization from the subsequent three models.

For the eighth model, we decided to try to improve our current reigning best model (model 6) by introducing dropout layers with 30% dropout rates after the flattening layer and after each max pooling layer. As reflected in Appendix A, this model resulted in the highest testing accuracy of the ten models that we constructed: 75.5%. Many other performance metrics performed well too with the testing precision for automobiles reaching 90% - a new high for our models. Another promising finding was that only 6 of the layer activation visualizations remained inactivated, which conveys that the channels were quite adept at identifying patterns in the images that were useful for classification. One final, great

finding was that the accuracy trendlines for the training and validation datasets across epochs during model fitting remained very close together, which suggests that the regularization was quite helpful (even if dropout can mask a bit of overfitting in these plots). While 75.5% computer vision testing accuracy may not be high enough for an autonomous vehicle in the real world, these results were still exciting and confirm that dropout regularization was very helpful for our modeling process.

The ninth model aimed to improve the eighth model by adding L1 regularization with a regularization parameter of 0.001 into the structure, but unfortunately, the results conveyed that this move decreased model performance yet again. Testing accuracy fell to 63.7% and noticeably more layer activation visualizations remained inactivated during this experiment than were inactivated in the previous experiment. One bright spot in these results was that the training, validation, and testing accuracies of 65.6%, 64.0%, and 63.7%, respectively, were quite close together, which suggests that this model was didn't require more regularization. Still, since the model performance dropped so much in this round, L1 regularization was not implemented from the subsequent experiment.

For the tenth experiment, we aimed to implement regularization by simplifying the model architecture in the flattened layer by reducing its number of nodes from 256 to 128, but unfortunately, this modification did not improve the model results compared to our reigning champion – model eight. The testing accuracy of 73.8% was our second highest recorded testing accuracy. One promising finding from Appendix C for this model was that validation accuracy actually outperformed training accuracy throughout most of the model fitting process, which suggests that we successfully implemented sufficient regularization. Another nice finding was that the confusion matrix in Appendix B conveyed that only 106 automobiles and trucks were misclassified as one another, which suggests that regularization had done a great deal to address this issue since the earlier models that had made roughly six times as many misclassifications of this type. While these results were all decently promising, notably, this model resulted in noticeably more inactive layer visualizations for the sampled images than model 8 had, which suggests that the channels were not identifying patterns as often as model 8 had been.

6. Conclusions

The ten DNN and CNN models constructed throughout our two phases of experimentation convey that deep learning techniques – particularly convolutional neural networks with several layers – can serve as powerful tools for classifying images into categories that are relevant for autonomous vehicles. Furthermore, the fact that the testing accuracy of these models ranged from 43.1% to 75.5% confirms that model architecture and regularization techniques play crucial roles in optimizing the performance of these models. If I were the data scientist responsible for advising an organization like Tesla about whether to proceed with implementing any of these computer vision algorithms, I'd likely encourage them to aim for higher than 75% testing accuracy since vehicles need much higher object recognition accuracy rates in order to properly protect passengers. More specifically, I would encourage them to experiment with more advanced techniques like regularization via data augmentation or building neural networks on top of successful, pre-trained computer vision models. That said, if the client needed to move forward with the implementation of one of these 10 models, I would advise that they select the eighth model since it achieved the highest level of predictive accuracy on the testing dataset.

References

- Elton, Charlotte. 2023. “Driverless cars: The dark side of autonomous vehicles that no one’s talking about.” *Euronews*. <https://www.euronews.com/green/2023/01/16/driverless-cars-the-dark-side-of-autonomous-vehicles-that-no-ones-talking-about#:~:text=There%20are%20currently%20more%20than,a%20computer%20consuming%20840%20watts>.
- Krizhevsky, Alex. 2017. “The CIFAR-10 dataset” University of Toronto.
<https://www.cs.toronto.edu/~kriz/cifar.html>
- Štetić, Filip. 2022. “The history of computer vision and the evolution of autonomous vehicles.” *Megatrend*. <https://www.megatrend.com/en/the-history-of-computer-vision-and-the-evolution-of-autonomous-vehicles/>
- Tanlq. 2022. “vit-base-patch16-224-in21k-finetuned-cifar10.” *Hugging Face*.
<https://huggingface.co/tanlq/vit-base-patch16-224-in21k-finetuned-cifar10>
- Wikipedia. 2023. “Self-driving car.” *Wikipedia*. https://en.wikipedia.org/wiki/Self-driving_car#Commercialization

Appendix A – Accuracy Results From Experiments

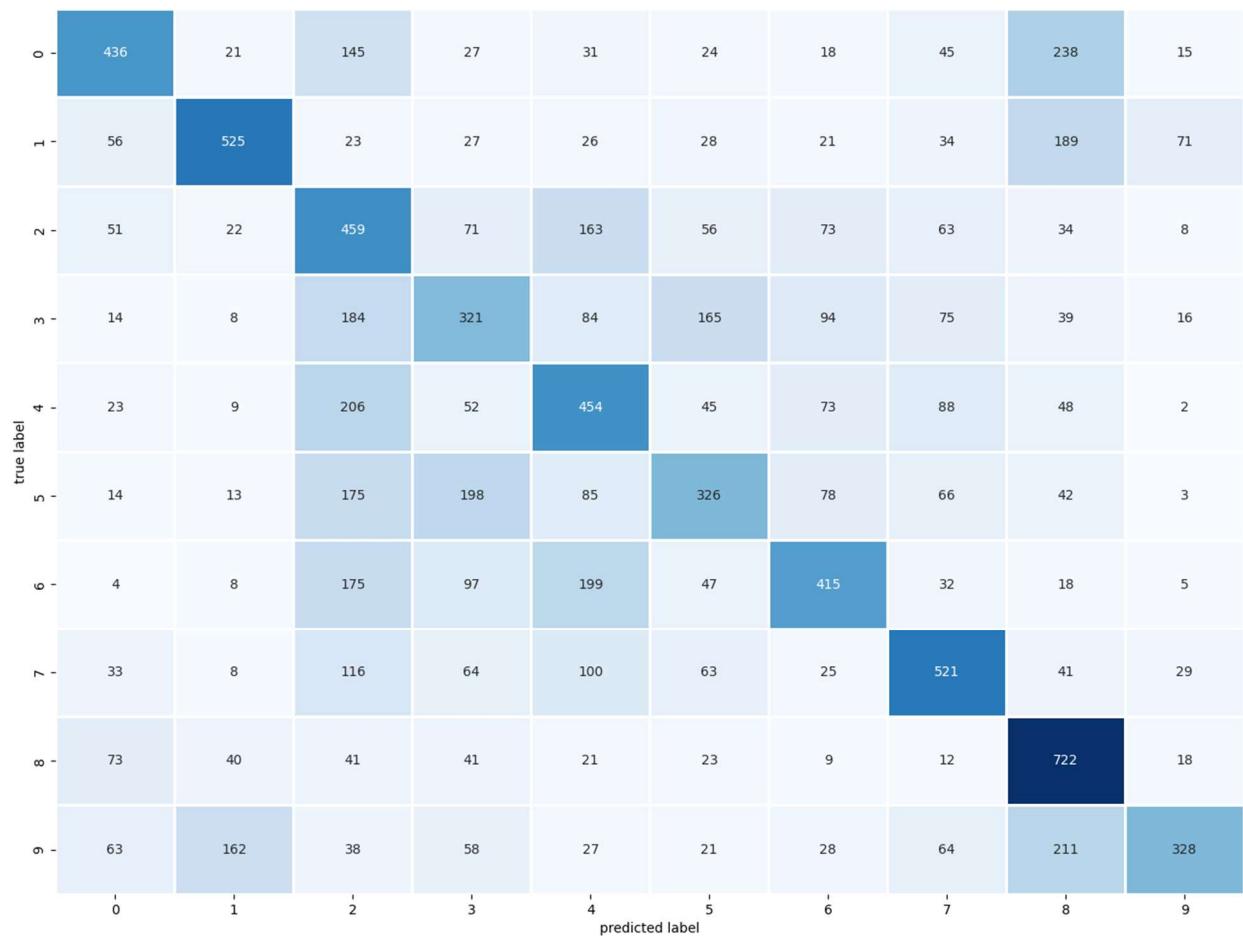
The table below displays the training, validation, and testing accuracy of each of the 10 models developed for classification of CIFAR-10 images.

Experiment Number	Model Type	Architecture Of Model Hidden Layers	Regularization Techniques Leveraged	Training Accuracy	Validation Accuracy	Testing Accuracy	Recommendation
1	DNN	• Dense Layer with 256 Nodes • Dense Layer with 64 Nodes	None	49.1%	44.9%	45.1%	Do Not Implement
2	DNN	• Dense Layer with 256 Nodes • Dense Layer with 128 Nodes • Dense Layer with 64 Nodes	None	45.7%	42.7%	43.1%	Do Not Implement
3	CNN	• 2D Convolutional Layer with 64 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 128 Filters • Max Pooling Layer with 2x2 Kernel • Flattened, Dense Layer with 384 Nodes	None	70.7%	55.2%	54.5%	Do Not Implement
4	CNN	• 2D Convolutional Layer with 32 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 64 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 128 Filters • Max Pooling Layer with 2x2 Kernel • Flattened, Dense Layer with 384 Nodes	None	71.3%	59.5%	58.5%	Do Not Implement
5	CNN	• 2D Convolutional Layer with 32 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 64 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 128 Filters • Max Pooling Layer with 2x2 Kernel • Flattened, Dense Layer with 256 Nodes	• Early Stopping • Simpler Architecture in Final Hidden Layer than Model 4	73.9%	63.6%	63.1%	Do Not Implement
6	CNN	• 2D Convolutional Layer with 32 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 64 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 128 Filters • Max Pooling Layer with 2x2 Kernel • Flattened, Dense Layer with 256 Nodes and Batch Normalization	• Early Stopping • Batch Normalization • Simpler Architecture in Final Hidden Layer than Model 4	82.5%	70.8%	69.4%	Do Not Implement
7	CNN	• 2D Convolutional Layer with 32 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 64 Filters • Max Pooling Layer with 2x2 Kernel • 2D Convolutional Layer with 128 Filters • Max Pooling Layer with 2x2 Kernel • Flattened, Dense Layer with 256 Nodes, Batch Normalization, and L2 Regularization	• Early Stopping • Batch Normalization • Simpler Architecture in Final Hidden Layer than Model 4 • L2 Regularization	70.7%	65.4%	66.0%	Do Not Implement
8	CNN	• 2D Convolutional Layer with 32 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • 2D Convolutional Layer with 64 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • 2D Convolutional Layer with 128 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • Flattened, Dense Layer with 256 Nodes, Batch Normalization, and Dropout	• Early Stopping • Batch Normalization • Simpler Architecture in Final Hidden Layer than Model 4 • Dropout	81.9%	76.1%	75.5%	Implement (if we must choose one of these models to implement)
9	CNN	• 2D Convolutional Layer with 32 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • 2D Convolutional Layer with 64 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • 2D Convolutional Layer with 128 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • Flattened, Dense Layer with 256 Nodes, Batch Normalization, and Dropout	• Early Stopping • Batch Normalization • Simpler Architecture in Final Hidden Layer than Model 4 • Dropout • L1 Regularization	65.6%	64.0%	63.7%	Do Not Implement
10	CNN	• 2D Convolutional Layer with 32 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • 2D Convolutional Layer with 64 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • 2D Convolutional Layer with 128 Filters • Max Pooling Layer with 2x2 Kernel and Dropout • Flattened, Dense Layer with 128 Nodes, Batch Normalization, and Dropout Regularization and ReLu Activation Function	• Early Stopping • Batch Normalization • Simpler Architecture in Final Hidden Layer than Previous Models (and different activation function) • Dropout	78.2%	75.2%	73.5%	Do Not Implement

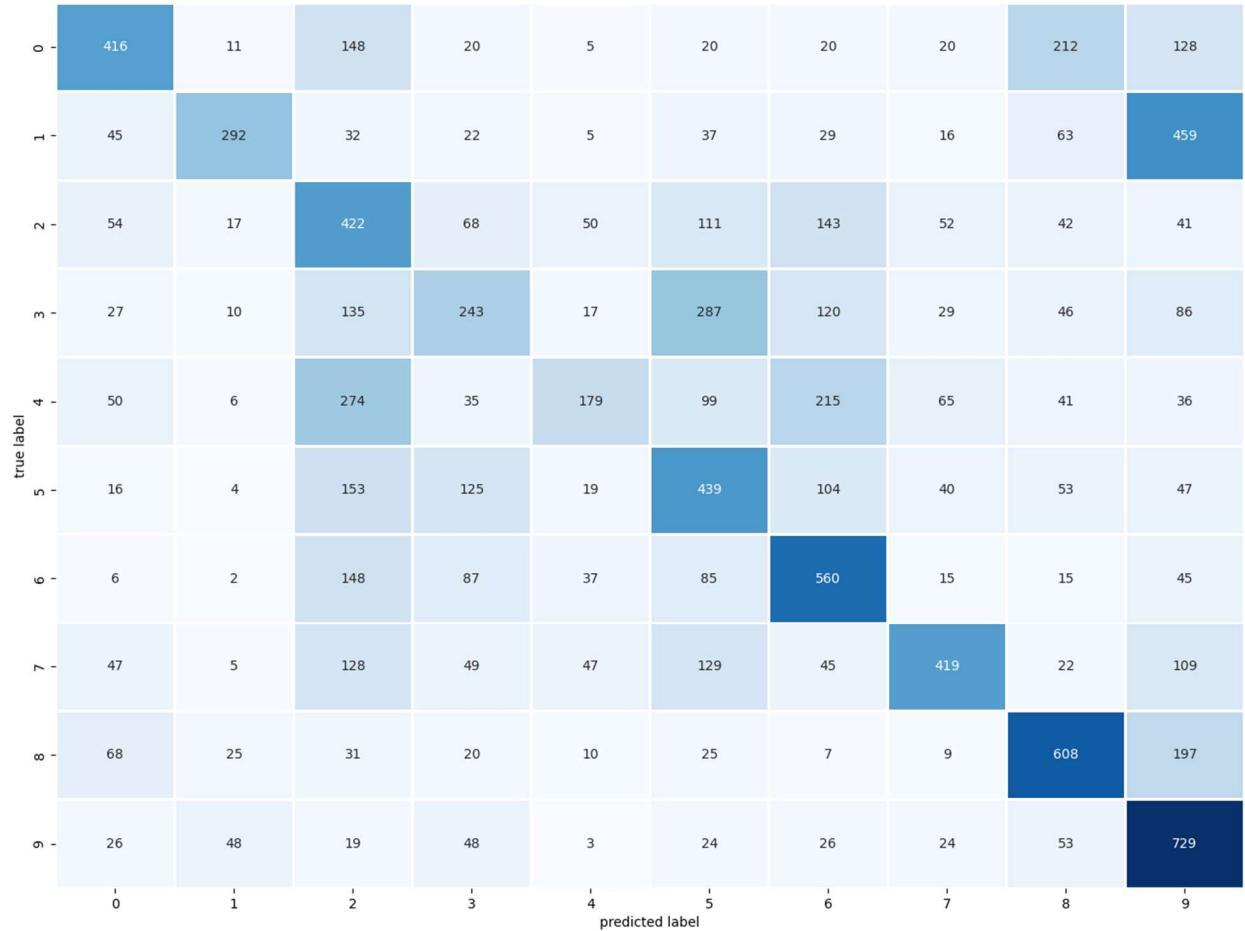
Appendix B – Confusion Matrices Resulting From Experiments

The images below display the confusion matrices resulting from the application of each CIFAR-10 classification model the testing dataset. For ease of interpretation, these confusion matrices are color coded as heat maps. Larger versions of these images and the code leveraged to generate these confusion matrices are available in Appendix D.

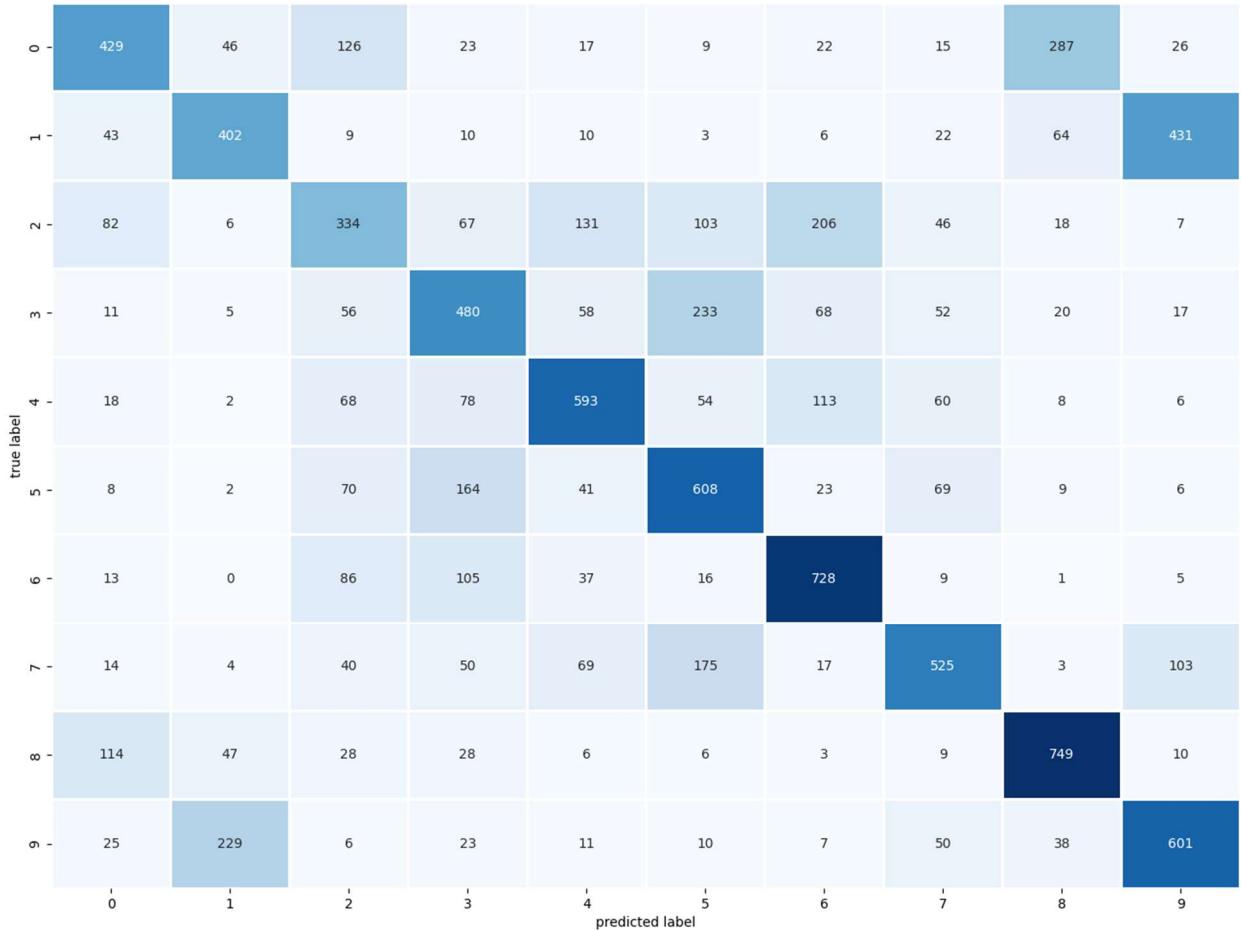
Experiment 1 – Confusion Matrix



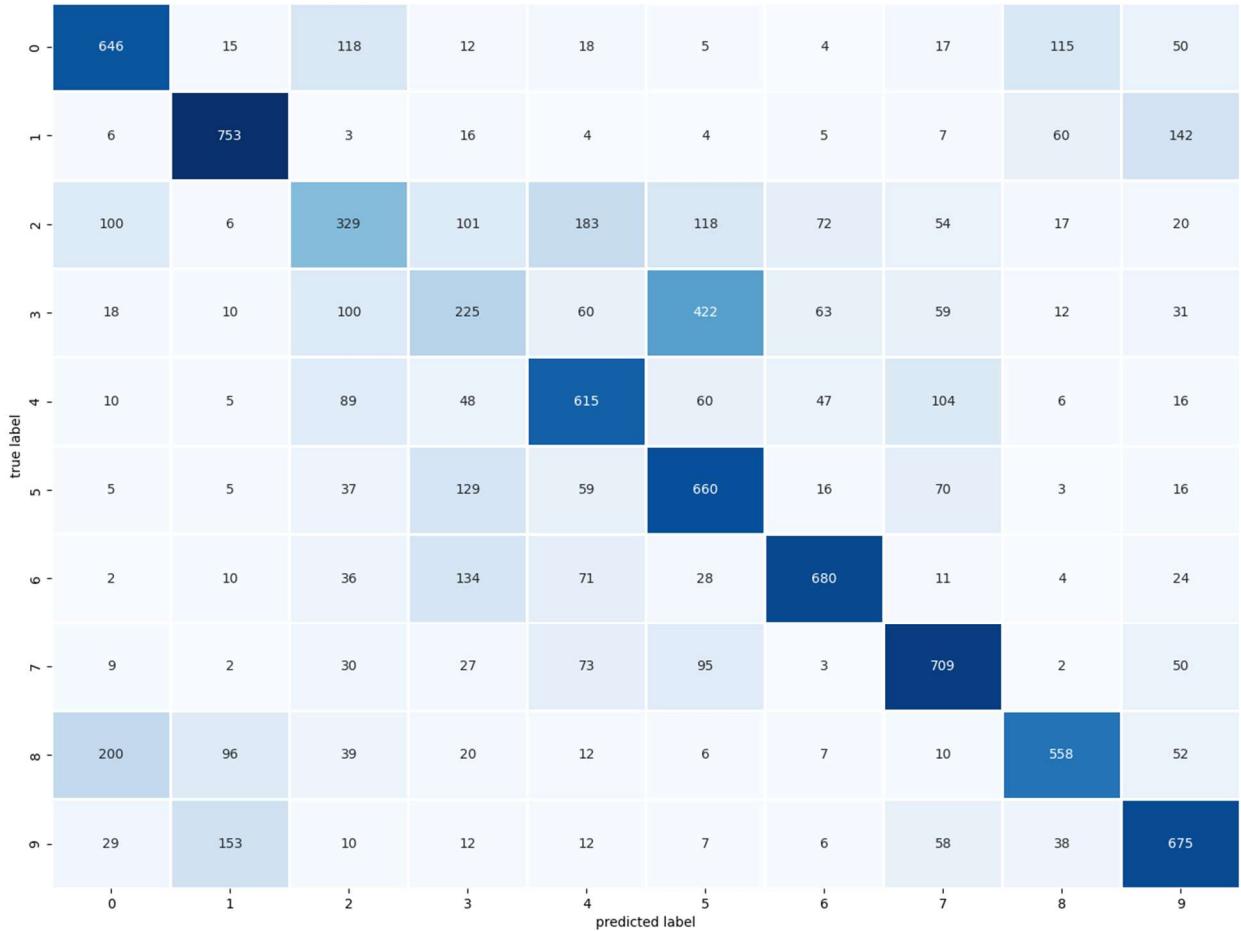
Experiment 2 – Confusion Matrix



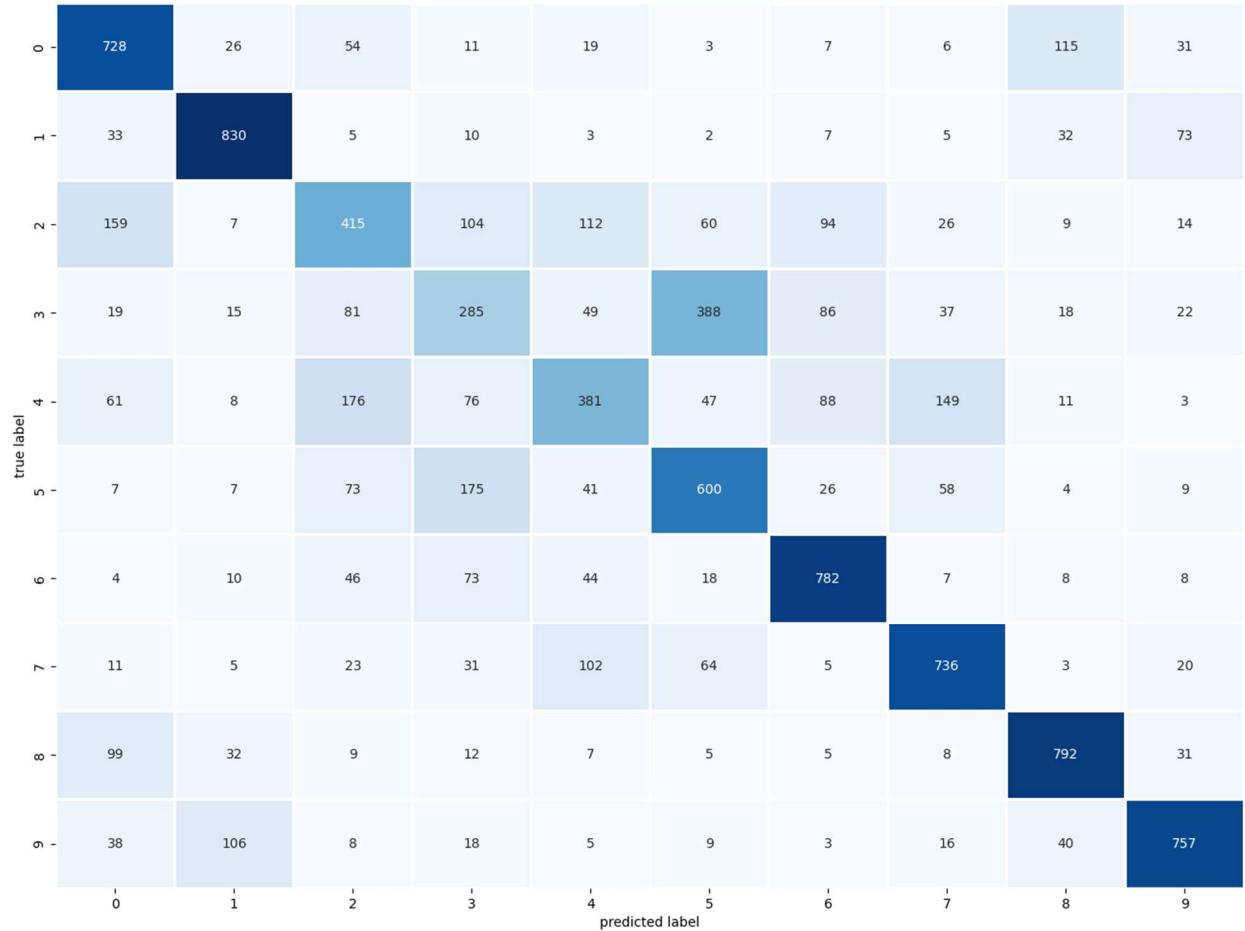
Experiment 3 – Confusion Matrix



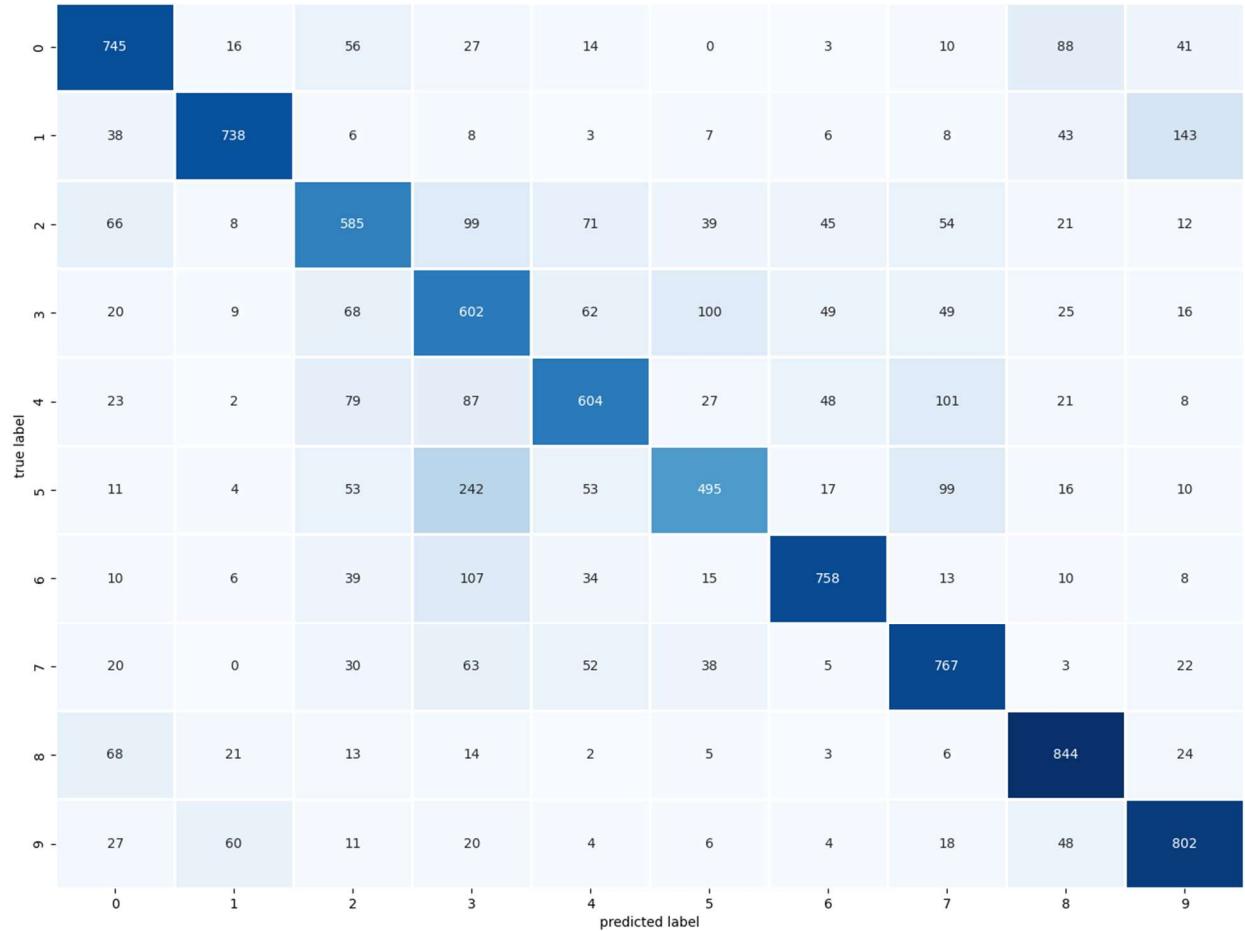
Experiment 4 – Confusion Matrix



Experiment 5 – Confusion Matrix



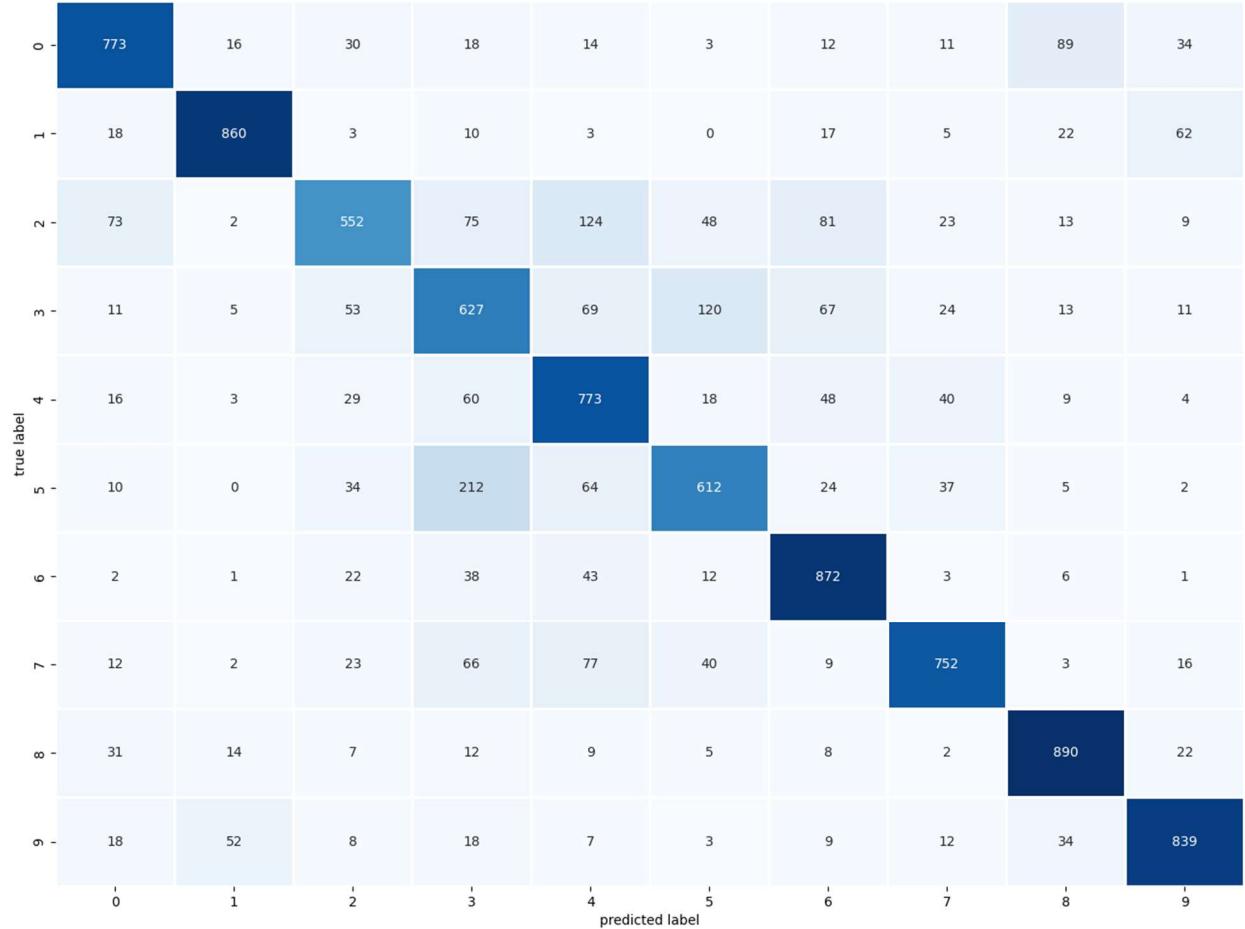
Experiment 6 – Confusion Matrix



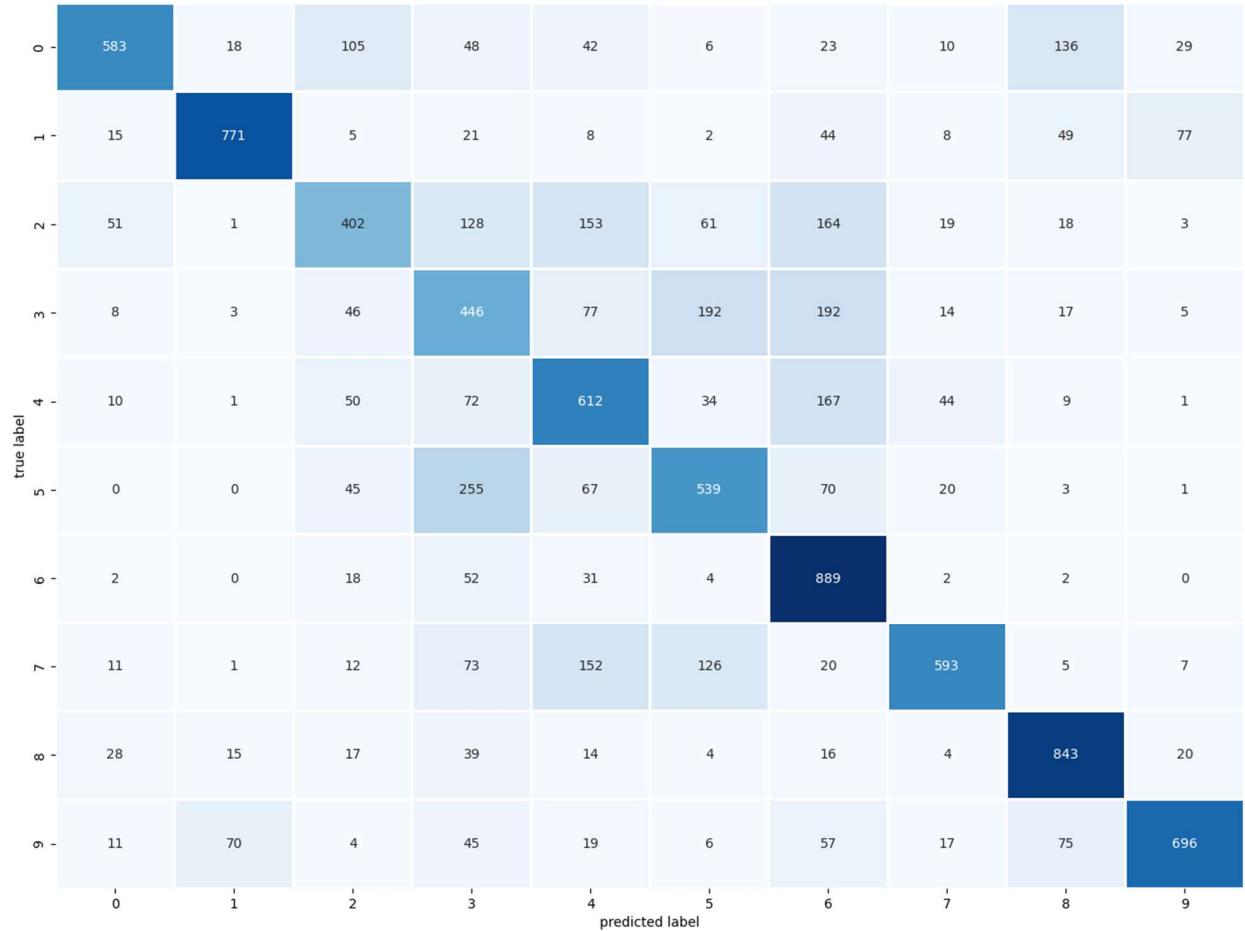
Experiment 7 – Confusion Matrix

true label	0	1	2	3	4	5	6	7	8	9	
	0	1	2	3	4	5	6	7	8	9	predicted label
0 -	633	58	58	7	35	4	7	13	154	31	
1 -	10	863	3	3	6	2	7	8	31	67	
2 -	51	24	555	27	194	30	25	58	25	11	
3 -	15	37	96	304	210	165	38	79	28	28	
4 -	9	17	43	11	803	12	11	69	20	5	
5 -	11	17	112	69	127	519	13	110	12	10	
6 -	7	13	67	35	219	22	598	12	13	14	
7 -	10	8	47	12	108	32	2	757	5	19	
8 -	22	59	11	3	17	3	1	3	851	30	
9 -	19	152	6	5	18	3	6	23	52	716	

Experiment 8 – Confusion Matrix



Experiment 9 – Confusion Matrix



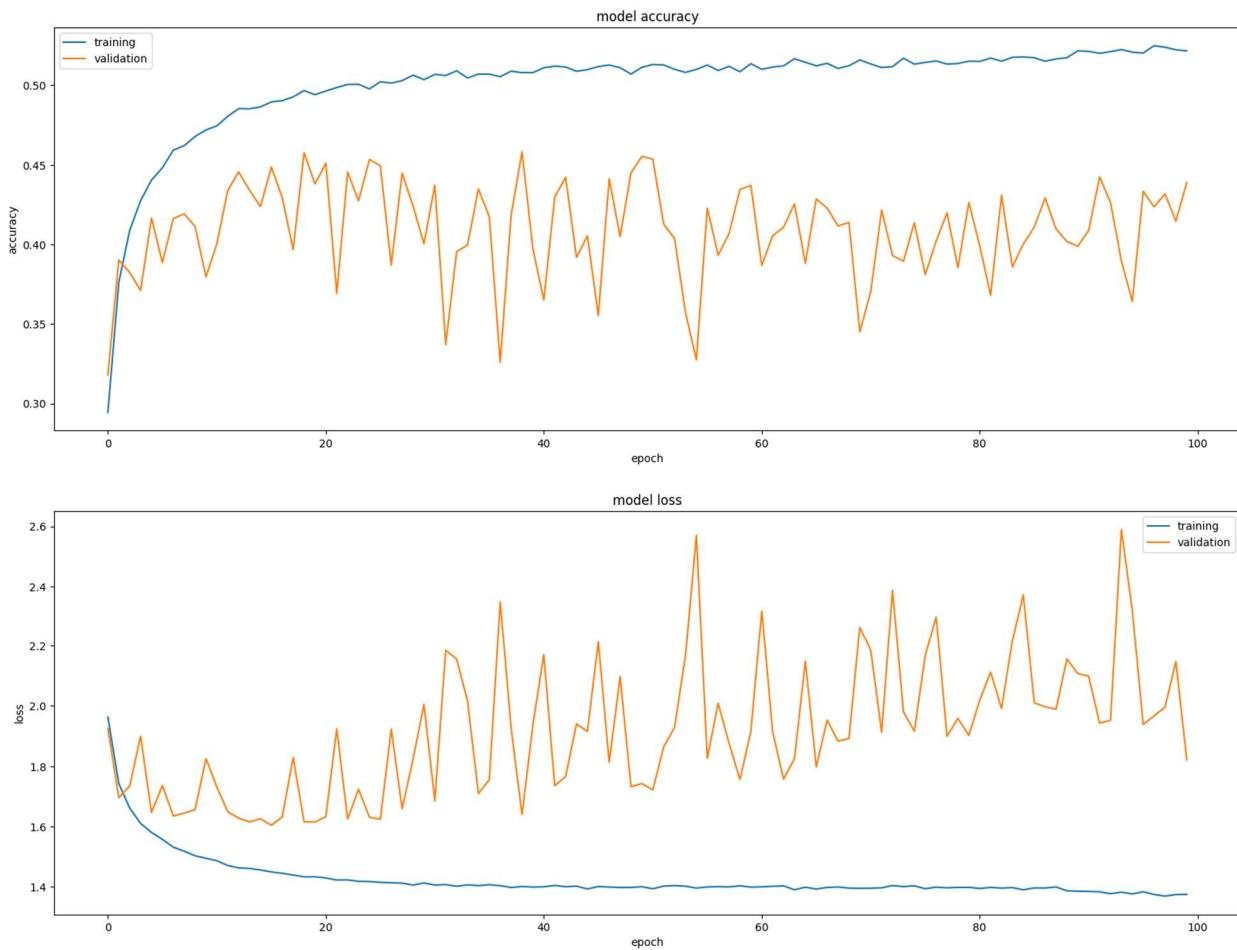
Experiment 10 – Confusion Matrix

	0	1	2	3	4	5	6	7	8	9	
true label	693	19	57	19	17	5	17	8	141	24	
	13	853	1	8	3	4	14	2	51	51	
	49	3	528	89	118	73	103	12	17	8	
	9	3	43	586	53	172	88	10	20	16	
	12	0	37	67	735	37	65	29	15	3	
	5	1	31	182	49	681	26	15	4	6	
	1	0	21	58	35	13	862	2	8	0	
	10	1	29	57	99	107	12	661	7	17	
	15	8	5	13	3	5	10	1	925	15	
	17	55	9	13	8	6	12	2	51	827	
	0	1	2	3	4	5	6	7	8	9	predicted label

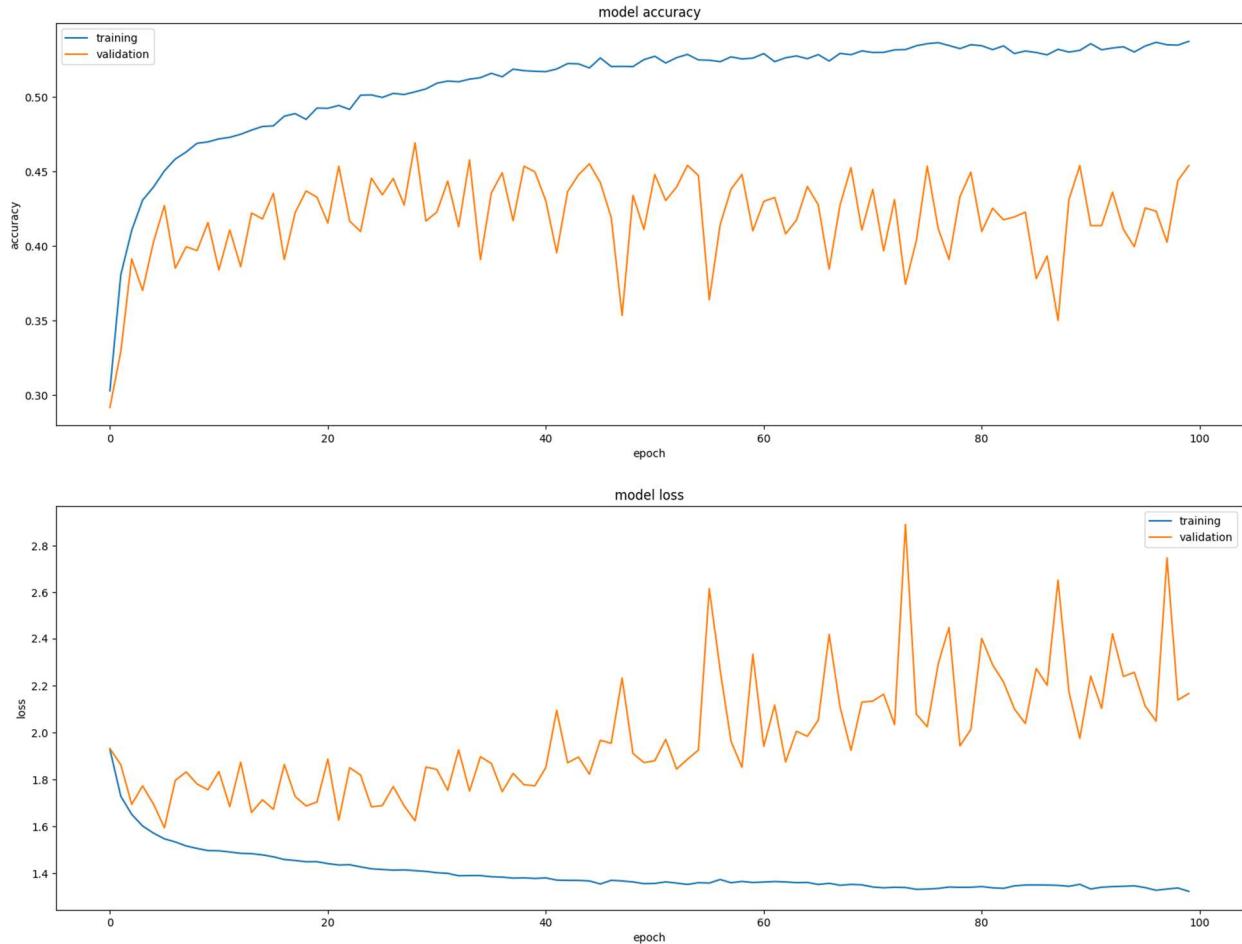
Appendix C – Accuracy and Loss Trends By Epoch Resulting From Experimental Model Fitting

The graphs below display the training and validation accuracies generated throughout each epoch of training each of the CIFAR-10 image classification models. Larger versions of these images and the code leveraged to generate these graphs are available in Appendix D.

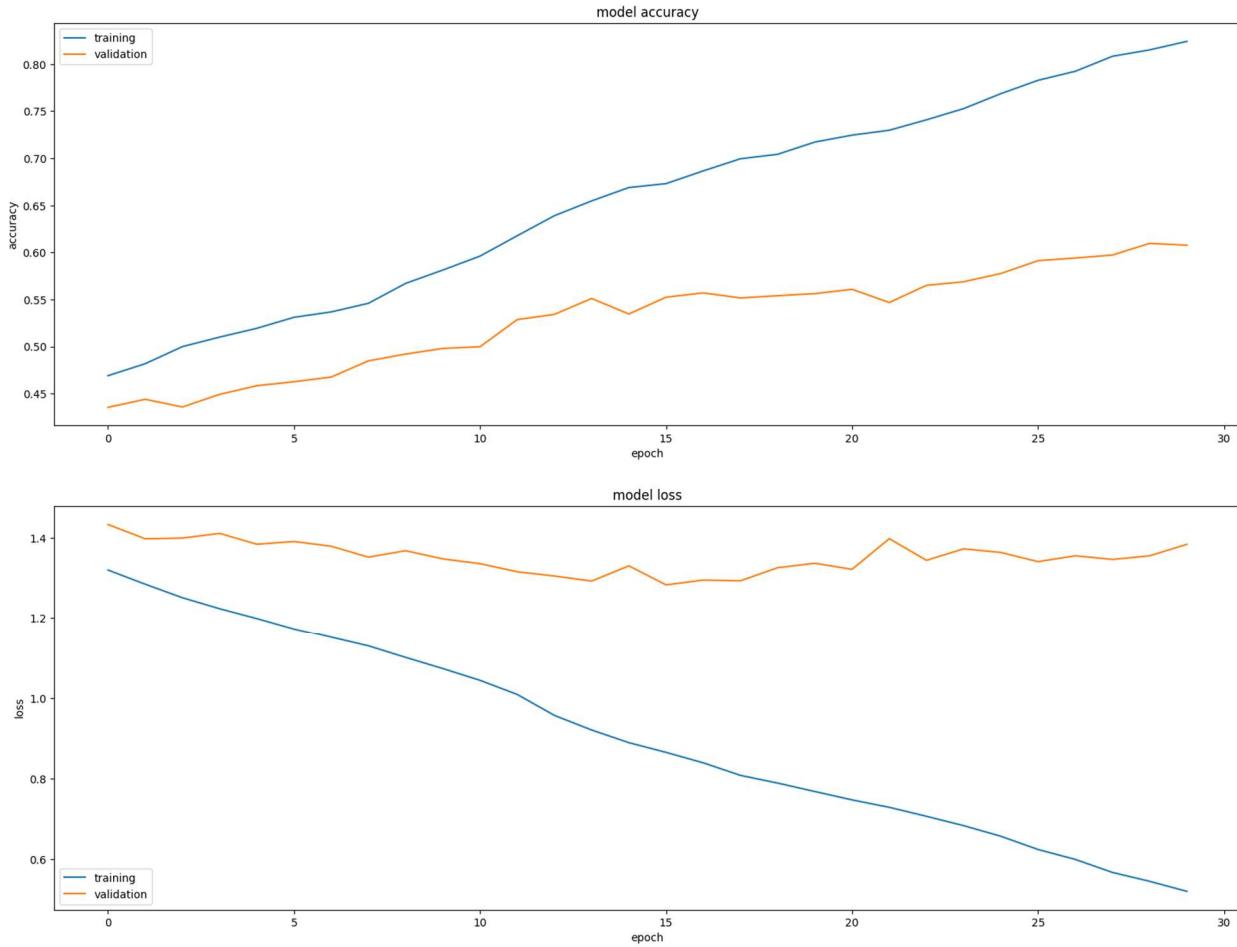
Experiment 1 – Accuracy and Loss Trends During Model Fitting



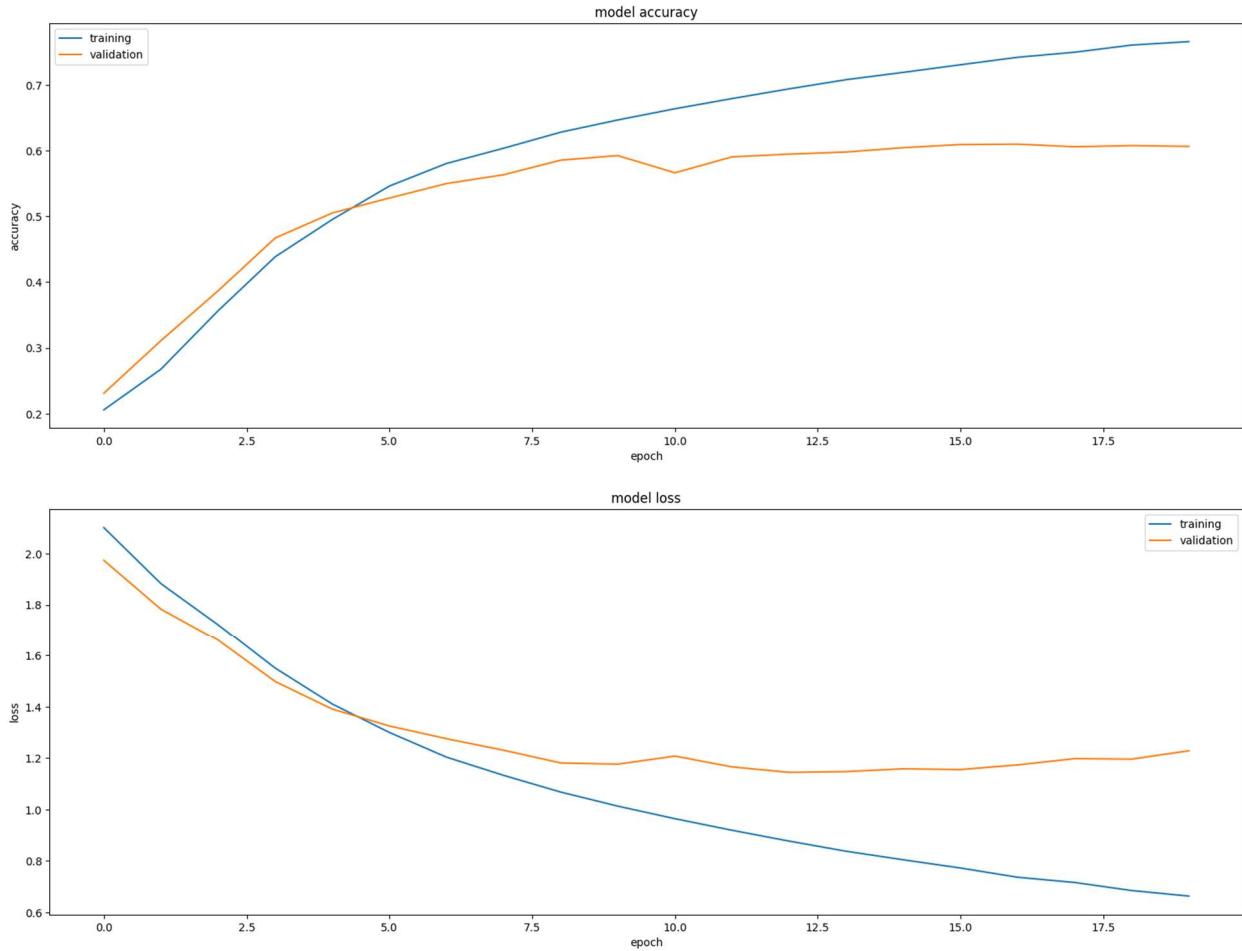
Experiment 2 – Accuracy and Loss Trends During Model Fitting



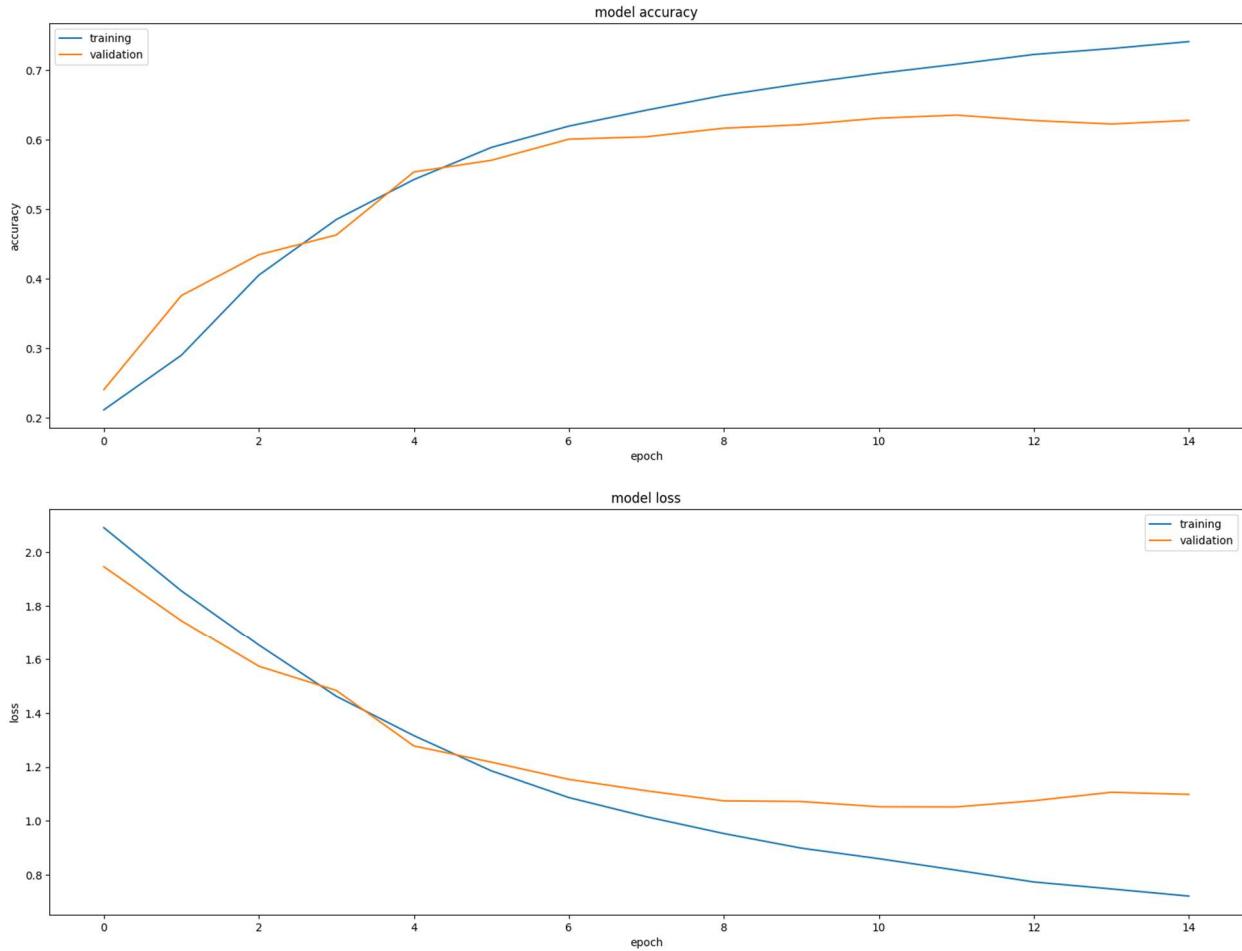
Experiment 3 – Accuracy and Loss Trends During Model Fitting



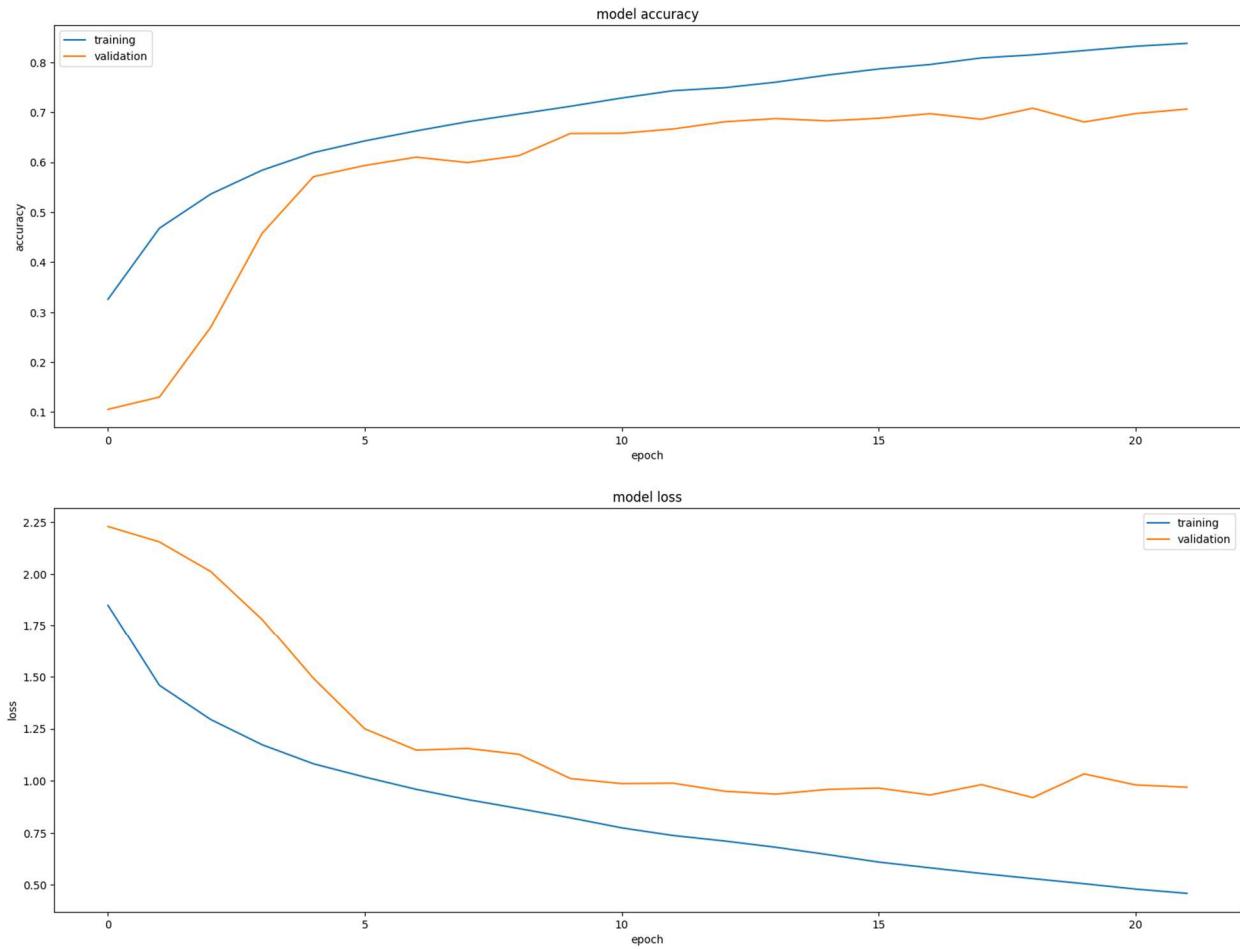
Experiment 4 – Accuracy and Loss Trends During Model Fitting



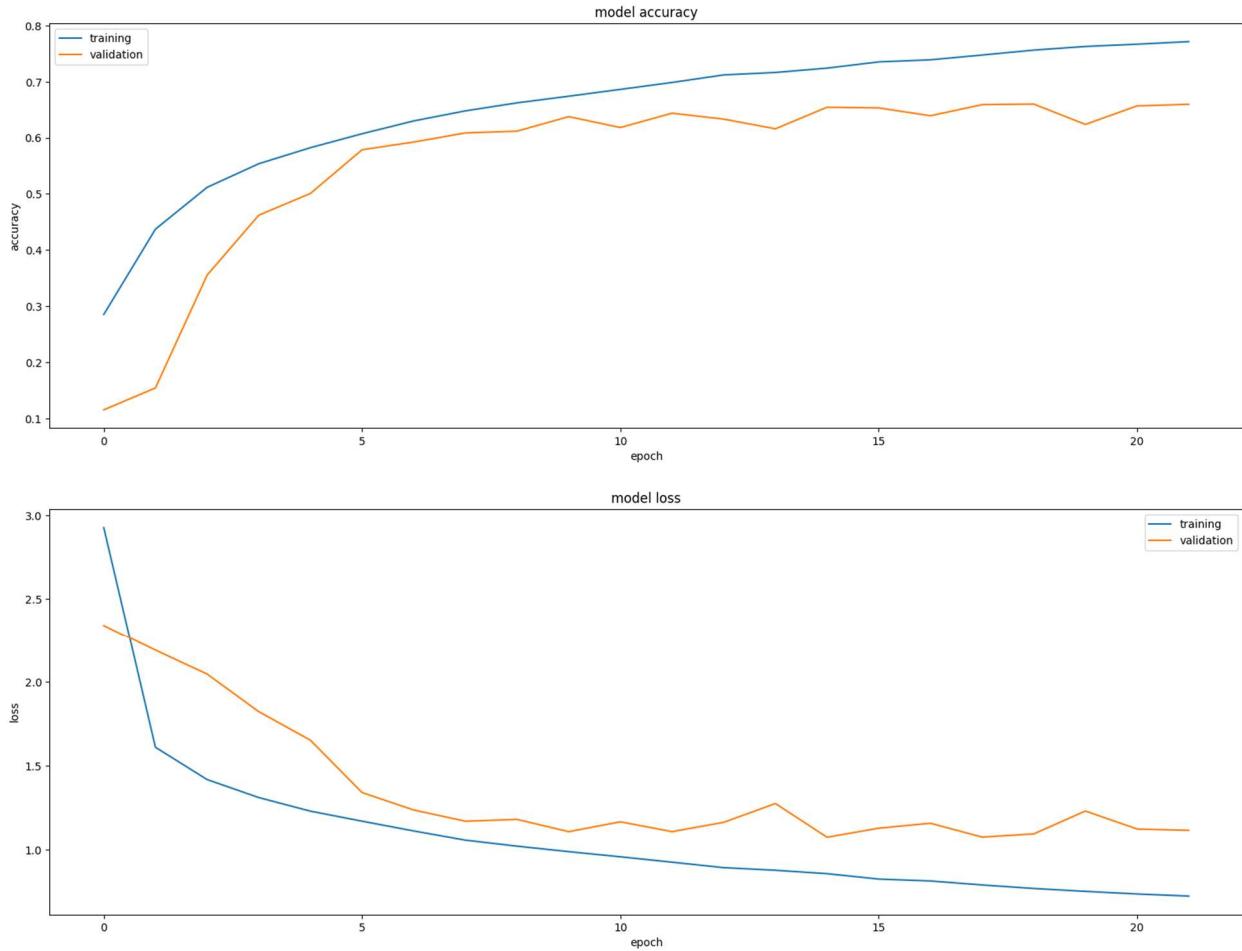
Experiment 5 – Accuracy and Loss Trends During Model Fitting



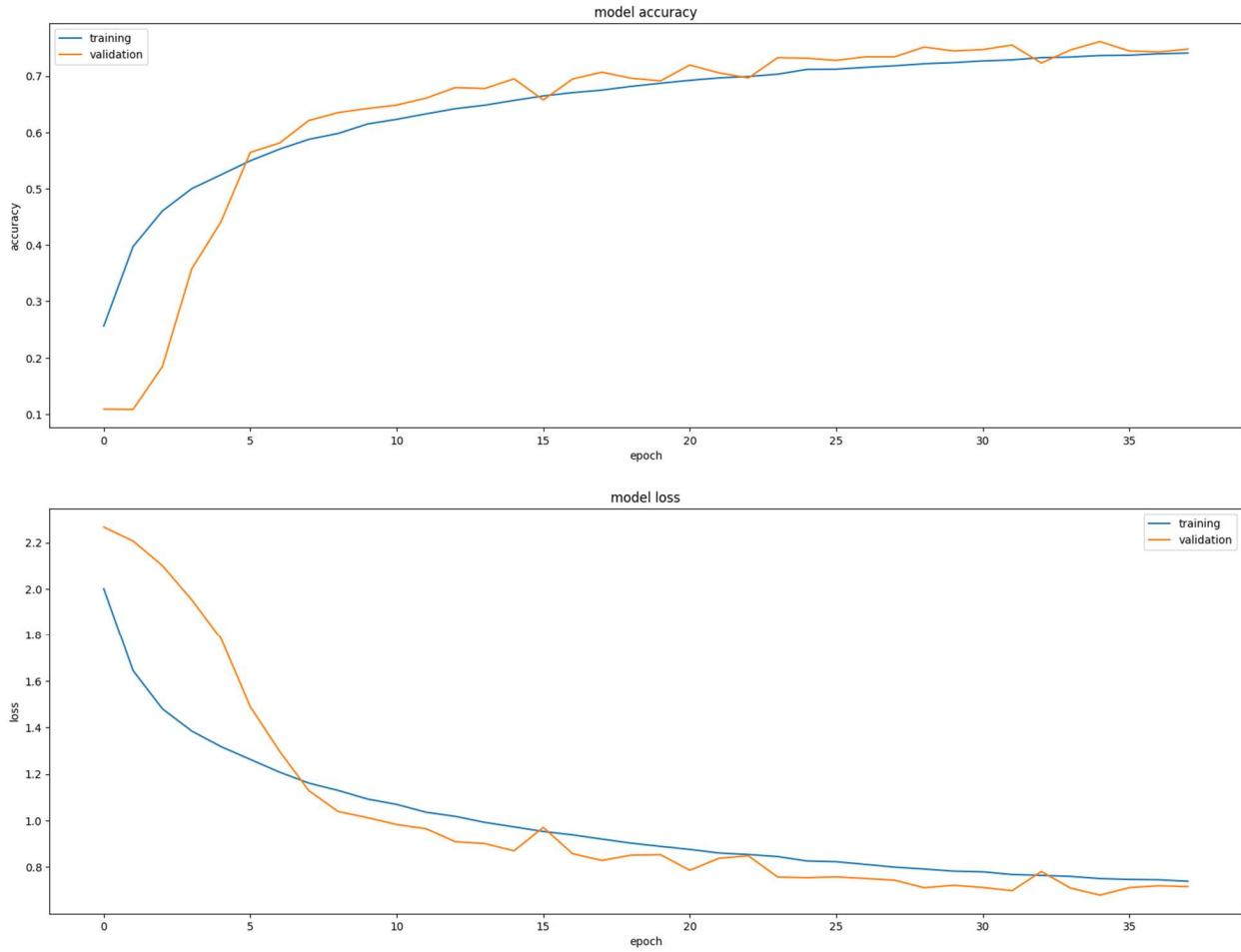
Experiment 6 – Accuracy and Loss Trends During Model Fitting



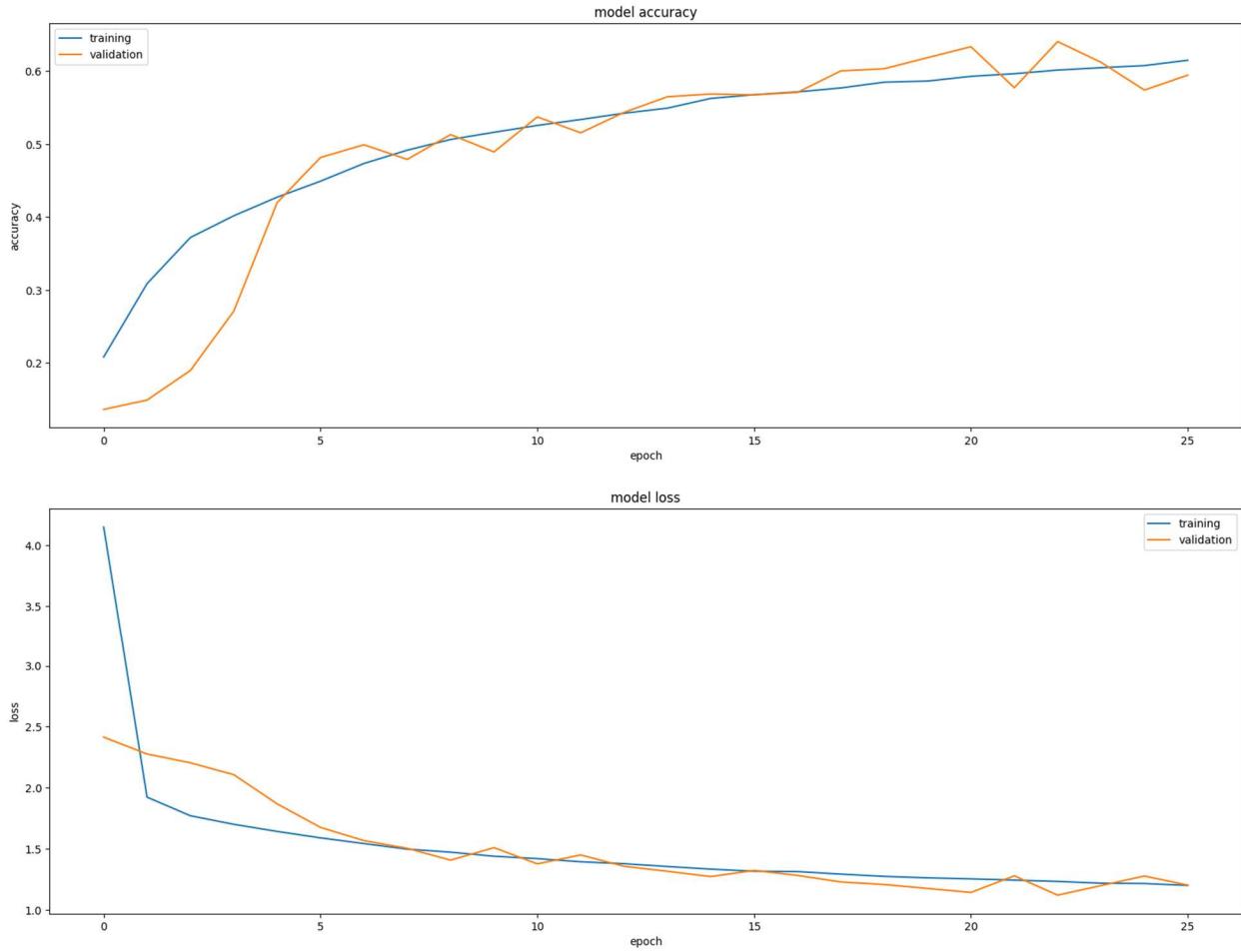
Experiment 7 – Accuracy and Loss Trends During Model Fitting



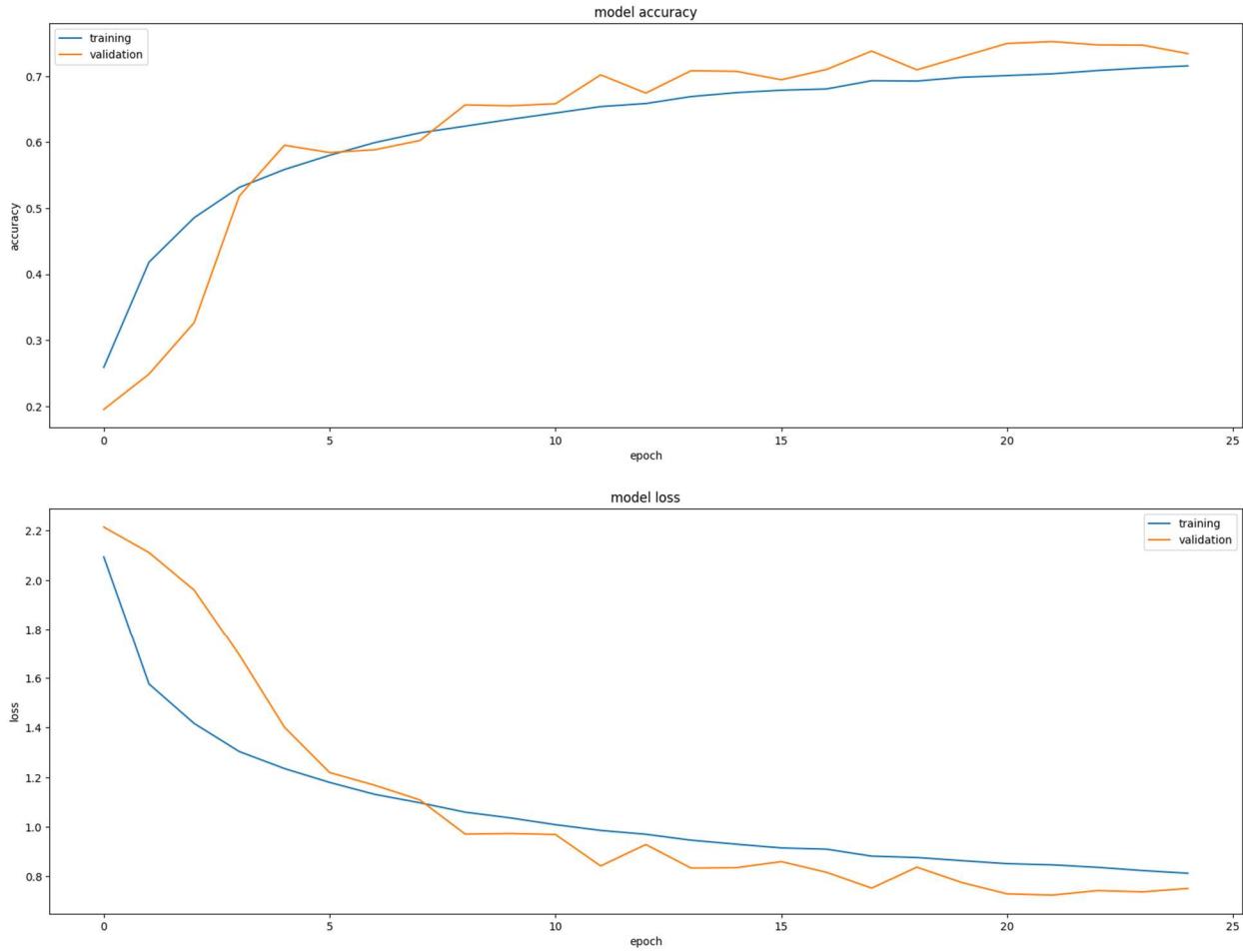
Experiment 8 – Accuracy and Loss Trends During Model Fitting



Experiment 9 – Accuracy and Loss Trends During Model Fitting



Experiment 10 – Accuracy and Loss Trends During Model Fitting



Appendix D - Supporting Python Code

Steve Desilets

October 22, 2023

1) Introduction

In this assignment, we will leverage the CIFAR-10 dataset train convolutional neural network (CNN) and deep neural network (DNN) models.

The CIFAR-10 dataset contains 60,000 training and 10,000 testing images that correspond to one of 10 categories:

1. Airplane
2. Automobile
3. Bird
4. Cat
5. Deer
6. Dog
7. Frog
8. Horse
9. Ship
10. Truck

We will aim to build the most accurate model possible and will examine the utility of the features extracted by each of these models during training.

The CIFAR-10 dataset

<https://www.cs.toronto.edu/~kriz/cifar.html>

1.1) Notebook Set-Up

Let's begin by importing the relevant packages.

```
In [ ]: import numpy as np
import pandas as pd
from packaging import version

from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error as MSE
from sklearn.model_selection import train_test_split
```

```
import matplotlib.pyplot as plt
import seaborn as sns

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import models, layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D, BatchNormalization, Dropout, Flatten
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.preprocessing import image
from tensorflow.keras.utils import to_categorical
import tensorflow.keras.backend as k
```

In []: `%matplotlib inline
np.set_printoptions(precision=3, suppress=True)`

Let's verify the version of Tensorflow.

In []: `print("This notebook requires TensorFlow 2.0 or above")
print("TensorFlow version: ", tf.__version__)
assert version.parse(tf.__version__).release[0] >=2`

This notebook requires TensorFlow 2.0 or above
TensorFlow version: 2.13.0

Let's mount to the Google Colab environment.

In []: `from google.colab import drive
drive.mount('/content/gdrive')`

Drive already mounted at /content/gdrive; to attempt to forcibly remount, call `drive.mount("/content/gdrive", force_remount=True)`.

Let's define functions that we can leverage for exploratory data analysis and for reporting.

In []: `def get_three_classes(x, y):
 def indices_of(class_id):
 indices, _ = np.where(y == float(class_id))
 return indices

 indices = np.concatenate([indices_of(0), indices_of(1), indices_of(2)], axis=0)

 x = x[indices]
 y = y[indices]

 count = x.shape[0]
 indices = np.random.choice(range(count), count, replace=False)

 x = x[indices]
 y = y[indices]

 y = tf.keras.utils.to_categorical(y)

 return x, y`

In []: `def show_random_examples(x, y, p):
 indices = np.random.choice(range(x.shape[0]), 10, replace=False)`

```

x = x[indices]
y = y[indices]
p = p[indices]

plt.figure(figsize=(10, 5))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x[i])
    plt.xticks([])
    plt.yticks([])
    col = 'green' if np.argmax(y[i]) == np.argmax(p[i]) else 'red'
    plt.xlabel(class_names_preview[np.argmax(p[i])], color=col)
plt.show()

```

```

In [ ]: def plot_history(history):
    losses = history.history['loss']
    accs = history.history['accuracy']
    val_losses = history.history['val_loss']
    val_accs = history.history['val_accuracy']
    epochs = len(losses)

    plt.figure(figsize=(16, 4))
    for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Accuracy'])):
        plt.subplot(1, 2, i + 1)
        plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
        plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
        plt.legend()
    plt.show()

def display_training_curves(training, validation, title, subplot):
    ax = plt.subplot(subplot)
    ax.plot(training)
    ax.plot(validation)
    ax.set_title('model ' + title)
    ax.set_ylabel(title)
    ax.set_xlabel('epoch')
    ax.legend(['training', 'validation'])

```

```

In [ ]: def print_validation_report(y_test, predictions):
    print("Classification Report")
    print(classification_report(y_test, predictions))
    print('Accuracy Score: {}'.format(accuracy_score(y_test, predictions)))
    print('Root Mean Square Error: {}'.format(np.sqrt(MSE(y_test, predictions))))

```

```

In [ ]: def plot_confusion_matrix(y_true, y_pred):
    mtx = confusion_matrix(y_true, y_pred)
    fig, ax = plt.subplots(figsize=(16,12))
    sns.heatmap(mtx, annot=True, fmt='d', linewidths=.75, cbar=False, ax=ax, cmap='Blues',
                # square=True,
                plt.ylabel('true label'),
                plt.xlabel('predicted label')

```

1.2) Exploratory Data Analysis and Data Pre-processing

Loading the CIFAR-10 Dataset

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images.

The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

```
In [ ]: (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
```

Let's conduct Exploratory Data Analysis on the imported data.

```
In [ ]: print('train_images:\t{}\n'.format(x_train.shape))
print('train_labels:\t{}\n'.format(y_train.shape))
print('test_images:\t{}\n'.format(x_test.shape))
print('test_labels:\t{}\n'.format(y_test.shape))
```

```
train_images:    (50000, 32, 32, 3)
train_labels:   (50000, 1)
test_images:     (10000, 32, 32, 3)
test_labels:    (10000, 1)
```

```
In [ ]: print("First ten labels training dataset:\n {}\n".format(y_train[0:10]))
print("This output the numeric label, need to convert to item description")
```

First ten labels training dataset:

```
[[6]
[9]
[9]
[4]
[1]
[1]
[2]
[7]
[8]
[3]]
```

This output the numeric label, need to convert to item description

Let's plot a subset of example images from this imported dataset.

```
In [ ]: (train_images, train_labels),(test_images, test_labels)= tf.keras.datasets.cifar10.lo
```

```
In [ ]: x_preview, y_preview = get_three_classes(train_images, train_labels)
x_preview, y_preview = get_three_classes(test_images, test_labels)
```

```
In [ ]: class_names_preview = ['aeroplane', 'car', 'bird']

show_random_examples(x_preview, y_preview, y_preview)
```



Let's preprocess the data prior to model development.

The labels are an array of integers, ranging from 0 to 9. These correspond to the class of clothing the image represents:

Label	Class_
0	airplane
1	automobile
2	bird
3	cat
4	deer
5	dog
6	frog
7	horse
8	ship
9	truck

```
In [ ]: class_names = ['airplane'
 , 'automobile'
 , 'bird'
 , 'cat'
 , 'deer'
 , 'dog'
 , 'frog'
 , 'horse'
 , 'ship'
 , 'truck']
```

Let's create a validation dataset.

```
In [ ]: x_train_split, x_valid_split, y_train_split, y_valid_split = train_test_split(x_train
, y_train
, test_size=0.2
, random_state=42
, shuffle=True)
```

```
In [ ]: print(x_train_split.shape, x_valid_split.shape, x_test.shape)
```

(45000, 32, 32, 3) (5000, 32, 32, 3) (10000, 32, 32, 3)

Let's rescale the data.

The images are 28x28 NumPy arrays, with pixel values ranging from 0 to 255.

1. Each element in each example is a pixel value
2. Pixel values range from 0 to 255
3. 0 = black
4. 255 = white

```
In [ ]: x_train_norm = x_train_split/255
x_valid_norm = x_valid_split/255
x_test_norm = x_test/255
```

```
In [ ]: y_train_split.shape
```

```
Out[ ]: (45000, 1)
```

2) Model 1 - Deep Neural Network with 2 Hidden Layers and No Regularization

2.1) Conduct Data Preprocessing For the Model

- Before we build our model, we need to prepare the data into the shape the network expected
- More specifically, we will convert the labels (integers 0 to 9) to 1D numpy arrays of shape (10,) with elements 0s and 1s.
- We also reshape the images from 3D arrays of shape (32, 32, 3) to 1D *float32* arrays of shape (3072,).

Let's apply one-hot coding to the labels.

We will change the way the labels are represented from numbers (0 to 9) to vectors (1D arrays) of shape (10,) with all the elements set to 0 except the one which the label belongs to - which will be set to 1. For example:

original label	one-hot encoded label
5	[0 0 0 0 1 0 0 0 0]

original label one-hot encoded label

7 [0 0 0 0 0 0 0 1 0 0]

1 [0 1 0 0 0 0 0 0 0 0]

```
In [ ]: y_train_encoded = to_categorical(y_train_split)
y_valid_encoded = to_categorical(y_valid_split)
y_test_encoded = to_categorical(y_test)

print("First ten entries of y_train_split:\n {}".format(y_train_split[0:10]))
print("First ten rows of one-hot y_train_encoded:\n {}".format(y_train_encoded[0:10,]))

print("First ten entries of y_valid_split:\n {}".format(y_valid_split[0:10]))
print("First ten rows of one-hot y_valid_encoded:\n {}".format(y_valid_encoded[0:10,]))

print("First ten entries of y_test:\n {}".format(y_test[0:10]))
print("First ten rows of one-hot y_test_encoded:\n {}".format(y_test_encoded[0:10,]))
```

First ten entries of y_train_split:

```
[[3]
 [1]
 [0]
 [3]
 [2]
 [4]
 [5]
 [9]
 [1]
 [5]]
```

First ten rows of one-hot y_train_encoded:

```
[[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

First ten entries of y_valid_split:

```
[[7]
 [8]
 [0]
 [6]
 [1]
 [6]
 [8]
 [0]
 [6]
 [5]]
```

First ten rows of one-hot y_valid_encoded:

```
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]
```

First ten entries of y_test:

```
[[3]
 [8]
 [8]
 [0]
 [6]
 [6]
 [1]
 [6]
 [3]
 [1]]
```

First ten rows of one-hot y_test_encoded:

```
[[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]]
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
```

In []: `print('y_train_encoded shape: ', y_train_encoded.shape)
print('y_valid_encoded shape:', y_valid_encoded.shape)
print('y_test_encoded shape: ', y_test_encoded.shape)`

```
y_train_encoded shape: (45000, 10)
y_valid_encoded shape: (5000, 10)
y_test_encoded shape: (10000, 10)
```

Reshape the images from shape (32, 32, 3) 2D arrays to shape (3072,) vectors (1D arrays).

In []: `# Before reshape:
print('x_train_norm:\t{}\t'.format(x_train_norm.shape))
print('x_valid_norm:\t{}\t'.format(x_valid_norm.shape))
print('x_test_norm:\t{}\t'.format(x_test_norm.shape))`

```
x_train_norm: (45000, 32, 32, 3)
x_valid_norm: (5000, 32, 32, 3)
x_test_norm: (10000, 32, 32, 3)
```

In []: `# Reshape the images:
x_train_reshaped = np.reshape(x_train_norm, (45000, 3072))
x_valid_reshaped = np.reshape(x_valid_norm, (5000, 3072))
x_test_reshaped = np.reshape(x_test_norm, (10000, 3072))

After reshape:
print('x_train_reshaped shape: ', x_train_reshaped.shape)
print('x_valid_reshaped shape: ', x_valid_reshaped.shape)
print('x_test_reshaped shape: ', x_test_reshaped.shape)`

```
x_train_reshaped shape: (45000, 3072)
x_valid_reshaped shape: (5000, 3072)
x_test_reshaped shape: (10000, 3072)
```

2.2) Build The Model

In []: `k.clear_session()

model = Sequential([
 Dense(input_shape=[3072], units=256, activation = tf.nn.relu),
 Dense(units=64, activation = tf.nn.relu),
 Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])`

In []: `model.summary()`

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	786688
dense_1 (Dense)	(None, 64)	16448
output_layer (Dense)	(None, 10)	650
<hr/>		
Total params: 803786 (3.07 MB)		
Trainable params: 803786 (3.07 MB)		
Non-trainable params: 0 (0.00 Byte)		

```
In [ ]: model.compile(optimizer='rmsprop',
                      loss = 'categorical_crossentropy',
                      metrics=['accuracy'])
```

```
In [ ]: #tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

history = model.fit(
    x_train_reshaped
    ,y_train_encoded
    ,epochs = 100
    ,validation_data=(x_valid_reshaped, y_valid_encoded)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("Model_1", save_best_only=True, save_)
```

```
Epoch 1/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.9626 - accuracy: 0.2944 - val_loss: 1.9252 - val_accuracy: 0.3178
Epoch 2/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.7421 - accuracy: 0.3762 - val_loss: 1.6956 - val_accuracy: 0.3902
Epoch 3/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.6611 - accuracy: 0.4088 - val_loss: 1.7344 - val_accuracy: 0.3824
Epoch 4/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.6102 - accuracy: 0.4275 - val_loss: 1.8993 - val_accuracy: 0.3710
Epoch 5/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.5804 - accuracy: 0.4404 - val_loss: 1.6466 - val_accuracy: 0.4166
Epoch 6/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.5572 - accuracy: 0.4482 - val_loss: 1.7360 - val_accuracy: 0.3886
Epoch 7/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.5316 - accuracy: 0.4593 - val_loss: 1.6350 - val_accuracy: 0.4162
Epoch 8/100
1407/1407 [=====] - 55s 39ms/step - loss: 1.5180 - accuracy: 0.4621 - val_loss: 1.6445 - val_accuracy: 0.4192
Epoch 9/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.5029 - accuracy: 0.4678 - val_loss: 1.6559 - val_accuracy: 0.4112
Epoch 10/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4946 - accuracy: 0.4720 - val_loss: 1.8251 - val_accuracy: 0.3796
Epoch 11/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4866 - accuracy: 0.4745 - val_loss: 1.7309 - val_accuracy: 0.4004
Epoch 12/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4707 - accuracy: 0.4806 - val_loss: 1.6490 - val_accuracy: 0.4340
Epoch 13/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4628 - accuracy: 0.4853 - val_loss: 1.6275 - val_accuracy: 0.4456
Epoch 14/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4608 - accuracy: 0.4852 - val_loss: 1.6153 - val_accuracy: 0.4340
Epoch 15/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4558 - accuracy: 0.4864 - val_loss: 1.6259 - val_accuracy: 0.4238
Epoch 16/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4495 - accuracy: 0.4896 - val_loss: 1.6039 - val_accuracy: 0.4486
Epoch 17/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4448 - accuracy: 0.4904 - val_loss: 1.6318 - val_accuracy: 0.4294
Epoch 18/100
1407/1407 [=====] - 12s 8ms/step - loss: 1.4392 - accuracy: 0.4927 - val_loss: 1.8289 - val_accuracy: 0.3968
Epoch 19/100
1407/1407 [=====] - 14s 10ms/step - loss: 1.4331 - accuracy: 0.4967 - val_loss: 1.6157 - val_accuracy: 0.4576
Epoch 20/100
1407/1407 [=====] - 12s 8ms/step - loss: 1.4333 - accuracy: 0.4941 - val_loss: 1.6148 - val_accuracy: 0.4380
```

```
Epoch 21/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4294 - accuracy: 0.4964 - val_loss: 1.6324 - val_accuracy: 0.4510
Epoch 22/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4226 - accuracy: 0.4986 - val_loss: 1.9236 - val_accuracy: 0.3692
Epoch 23/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4229 - accuracy: 0.5006 - val_loss: 1.6249 - val_accuracy: 0.4454
Epoch 24/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4182 - accuracy: 0.5007 - val_loss: 1.7234 - val_accuracy: 0.4274
Epoch 25/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4176 - accuracy: 0.4978 - val_loss: 1.6307 - val_accuracy: 0.4534
Epoch 26/100
1407/1407 [=====] - 12s 8ms/step - loss: 1.4147 - accuracy: 0.5022 - val_loss: 1.6239 - val_accuracy: 0.4494
Epoch 27/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4134 - accuracy: 0.5014 - val_loss: 1.9228 - val_accuracy: 0.3870
Epoch 28/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4123 - accuracy: 0.5030 - val_loss: 1.6592 - val_accuracy: 0.4448
Epoch 29/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.4058 - accuracy: 0.5064 - val_loss: 1.8232 - val_accuracy: 0.4242
Epoch 30/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4127 - accuracy: 0.5036 - val_loss: 2.0049 - val_accuracy: 0.4004
Epoch 31/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4060 - accuracy: 0.5069 - val_loss: 1.6854 - val_accuracy: 0.4372
Epoch 32/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4073 - accuracy: 0.5062 - val_loss: 2.1846 - val_accuracy: 0.3370
Epoch 33/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4018 - accuracy: 0.5091 - val_loss: 2.1552 - val_accuracy: 0.3954
Epoch 34/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.4065 - accuracy: 0.5046 - val_loss: 2.0165 - val_accuracy: 0.3996
Epoch 35/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.4041 - accuracy: 0.5071 - val_loss: 1.7094 - val_accuracy: 0.4348
Epoch 36/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.4071 - accuracy: 0.5070 - val_loss: 1.7555 - val_accuracy: 0.4174
Epoch 37/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.4037 - accuracy: 0.5054 - val_loss: 2.3482 - val_accuracy: 0.3258
Epoch 38/100
1407/1407 [=====] - 11s 7ms/step - loss: 1.3979 - accuracy: 0.5089 - val_loss: 1.9269 - val_accuracy: 0.4184
Epoch 39/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.4010 - accuracy: 0.5080 - val_loss: 1.6397 - val_accuracy: 0.4582
Epoch 40/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.3991 - accuracy: 0.5080 - val_loss: 1.9378 - val_accuracy: 0.3972
```

```
Epoch 41/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.4000 - accuracy: 0.5110 - val_loss: 2.1700 - val_accuracy: 0.3650
Epoch 42/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4047 - accuracy: 0.5121 - val_loss: 1.7353 - val_accuracy: 0.4296
Epoch 43/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4002 - accuracy: 0.5116 - val_loss: 1.7654 - val_accuracy: 0.4422
Epoch 44/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4022 - accuracy: 0.5088 - val_loss: 1.9403 - val_accuracy: 0.3918
Epoch 45/100
1407/1407 [=====] - 12s 8ms/step - loss: 1.3929 - accuracy: 0.5099 - val_loss: 1.9154 - val_accuracy: 0.4054
Epoch 46/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.4010 - accuracy: 0.5118 - val_loss: 2.2125 - val_accuracy: 0.3552
Epoch 47/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.3992 - accuracy: 0.5128 - val_loss: 1.8127 - val_accuracy: 0.4412
Epoch 48/100
1407/1407 [=====] - 12s 9ms/step - loss: 1.3978 - accuracy: 0.5111 - val_loss: 2.0969 - val_accuracy: 0.4050
Epoch 49/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3979 - accuracy: 0.5071 - val_loss: 1.7322 - val_accuracy: 0.4450
Epoch 50/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.4003 - accuracy: 0.5113 - val_loss: 1.7424 - val_accuracy: 0.4554
Epoch 51/100
1407/1407 [=====] - 12s 8ms/step - loss: 1.3936 - accuracy: 0.5131 - val_loss: 1.7215 - val_accuracy: 0.4536
Epoch 52/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4023 - accuracy: 0.5128 - val_loss: 1.8638 - val_accuracy: 0.4128
Epoch 53/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.4042 - accuracy: 0.5101 - val_loss: 1.9293 - val_accuracy: 0.4036
Epoch 54/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4019 - accuracy: 0.5081 - val_loss: 2.1675 - val_accuracy: 0.3572
Epoch 55/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.3958 - accuracy: 0.5100 - val_loss: 2.5703 - val_accuracy: 0.3274
Epoch 56/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3995 - accuracy: 0.5128 - val_loss: 1.8266 - val_accuracy: 0.4228
Epoch 57/100
1407/1407 [=====] - 8s 5ms/step - loss: 1.4004 - accuracy: 0.5093 - val_loss: 2.0085 - val_accuracy: 0.3932
Epoch 58/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3996 - accuracy: 0.5119 - val_loss: 1.8758 - val_accuracy: 0.4068
Epoch 59/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4033 - accuracy: 0.5086 - val_loss: 1.7555 - val_accuracy: 0.4346
Epoch 60/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3988 - accuracy: 0.5136 - val_loss: 1.9151 - val_accuracy: 0.4370
```

```
Epoch 61/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3999 - accuracy: 0.5101 - val_loss: 2.3178 - val_accuracy: 0.3868
Epoch 62/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.4016 - accuracy: 0.5115 - val_loss: 1.9132 - val_accuracy: 0.4054
Epoch 63/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4031 - accuracy: 0.5123 - val_loss: 1.7567 - val_accuracy: 0.4108
Epoch 64/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3902 - accuracy: 0.5167 - val_loss: 1.8255 - val_accuracy: 0.4254
Epoch 65/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3987 - accuracy: 0.5145 - val_loss: 2.1478 - val_accuracy: 0.3882
Epoch 66/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3927 - accuracy: 0.5122 - val_loss: 1.7980 - val_accuracy: 0.4286
Epoch 67/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3980 - accuracy: 0.5138 - val_loss: 1.9526 - val_accuracy: 0.4228
Epoch 68/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3994 - accuracy: 0.5106 - val_loss: 1.8831 - val_accuracy: 0.4116
Epoch 69/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3957 - accuracy: 0.5123 - val_loss: 1.8919 - val_accuracy: 0.4138
Epoch 70/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3948 - accuracy: 0.5160 - val_loss: 2.2638 - val_accuracy: 0.3450
Epoch 71/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3953 - accuracy: 0.5134 - val_loss: 2.1845 - val_accuracy: 0.3704
Epoch 72/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3967 - accuracy: 0.5112 - val_loss: 1.9130 - val_accuracy: 0.4216
Epoch 73/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4043 - accuracy: 0.5117 - val_loss: 2.3873 - val_accuracy: 0.3930
Epoch 74/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4006 - accuracy: 0.5171 - val_loss: 1.9800 - val_accuracy: 0.3894
Epoch 75/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4034 - accuracy: 0.5133 - val_loss: 1.9161 - val_accuracy: 0.4136
Epoch 76/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3936 - accuracy: 0.5144 - val_loss: 2.1665 - val_accuracy: 0.3810
Epoch 77/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3990 - accuracy: 0.5154 - val_loss: 2.2984 - val_accuracy: 0.4020
Epoch 78/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3967 - accuracy: 0.5134 - val_loss: 1.8994 - val_accuracy: 0.4198
Epoch 79/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3982 - accuracy: 0.5137 - val_loss: 1.9584 - val_accuracy: 0.3854
Epoch 80/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3983 - accuracy: 0.5152 - val_loss: 1.9020 - val_accuracy: 0.4264
```

```
Epoch 81/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3944 - accuracy: 0.5151 - val_loss: 2.0188 - val_accuracy: 0.3992
Epoch 82/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3980 - accuracy: 0.5171 - val_loss: 2.1119 - val_accuracy: 0.3680
Epoch 83/100
1407/1407 [=====] - 8s 5ms/step - loss: 1.3958 - accuracy: 0.5152 - val_loss: 1.9915 - val_accuracy: 0.4308
Epoch 84/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3975 - accuracy: 0.5176 - val_loss: 2.2185 - val_accuracy: 0.3858
Epoch 85/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3902 - accuracy: 0.5179 - val_loss: 2.3723 - val_accuracy: 0.4002
Epoch 86/100
1407/1407 [=====] - 8s 5ms/step - loss: 1.3963 - accuracy: 0.5174 - val_loss: 2.0101 - val_accuracy: 0.4108
Epoch 87/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3960 - accuracy: 0.5152 - val_loss: 1.9974 - val_accuracy: 0.4292
Epoch 88/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3996 - accuracy: 0.5166 - val_loss: 1.9888 - val_accuracy: 0.4098
Epoch 89/100
1407/1407 [=====] - 8s 5ms/step - loss: 1.3869 - accuracy: 0.5174 - val_loss: 2.1557 - val_accuracy: 0.4018
Epoch 90/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3853 - accuracy: 0.5217 - val_loss: 2.1069 - val_accuracy: 0.3988
Epoch 91/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3847 - accuracy: 0.5214 - val_loss: 2.0983 - val_accuracy: 0.4088
Epoch 92/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3835 - accuracy: 0.5201 - val_loss: 1.9426 - val_accuracy: 0.4424
Epoch 93/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3772 - accuracy: 0.5212 - val_loss: 1.9524 - val_accuracy: 0.4264
Epoch 94/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3823 - accuracy: 0.5225 - val_loss: 2.5892 - val_accuracy: 0.3896
Epoch 95/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3764 - accuracy: 0.5209 - val_loss: 2.3194 - val_accuracy: 0.3642
Epoch 96/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3838 - accuracy: 0.5203 - val_loss: 1.9383 - val_accuracy: 0.4334
Epoch 97/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3745 - accuracy: 0.5249 - val_loss: 1.9671 - val_accuracy: 0.4236
Epoch 98/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3692 - accuracy: 0.5240 - val_loss: 1.9957 - val_accuracy: 0.4318
Epoch 99/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3743 - accuracy: 0.5224 - val_loss: 2.1471 - val_accuracy: 0.4146
Epoch 100/100
1407/1407 [=====] - 8s 5ms/step - loss: 1.3749 - accuracy: 0.5217 - val_loss: 1.8211 - val_accuracy: 0.4390
```

2.2) Evaluate Model Performance

Now that we've fit our model, let's apply the model to the test dataset and subsequently evaluate its performance.

```
In [ ]: model = tf.keras.models.load_model("Model_1")
print(f"Training accuracy: {model.evaluate(x_train_reshaped, y_train_encoded)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_reshaped, y_valid_encoded)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_reshaped, y_test_encoded)[1]:.3f}")

1407/1407 [=====] - 3s 2ms/step - loss: 1.4308 - accuracy: 0.4911
Training accuracy: 0.491
157/157 [=====] - 0s 2ms/step - loss: 1.6039 - accuracy: 0.486
Validation accuracy: 0.449
313/313 [=====] - 1s 2ms/step - loss: 1.5690 - accuracy: 0.4507
Test accuracy: 0.451
```

```
In [ ]: # loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

```
In [ ]: preds = model.predict(x_test_reshaped)
print('shape of preds: ', preds.shape)

313/313 [=====] - 1s 3ms/step
shape of preds: (10000, 10)
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

```
In [ ]: plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color=col)
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

```
In [ ]: history_dict = history.history
history_dict.keys()
```

```
Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

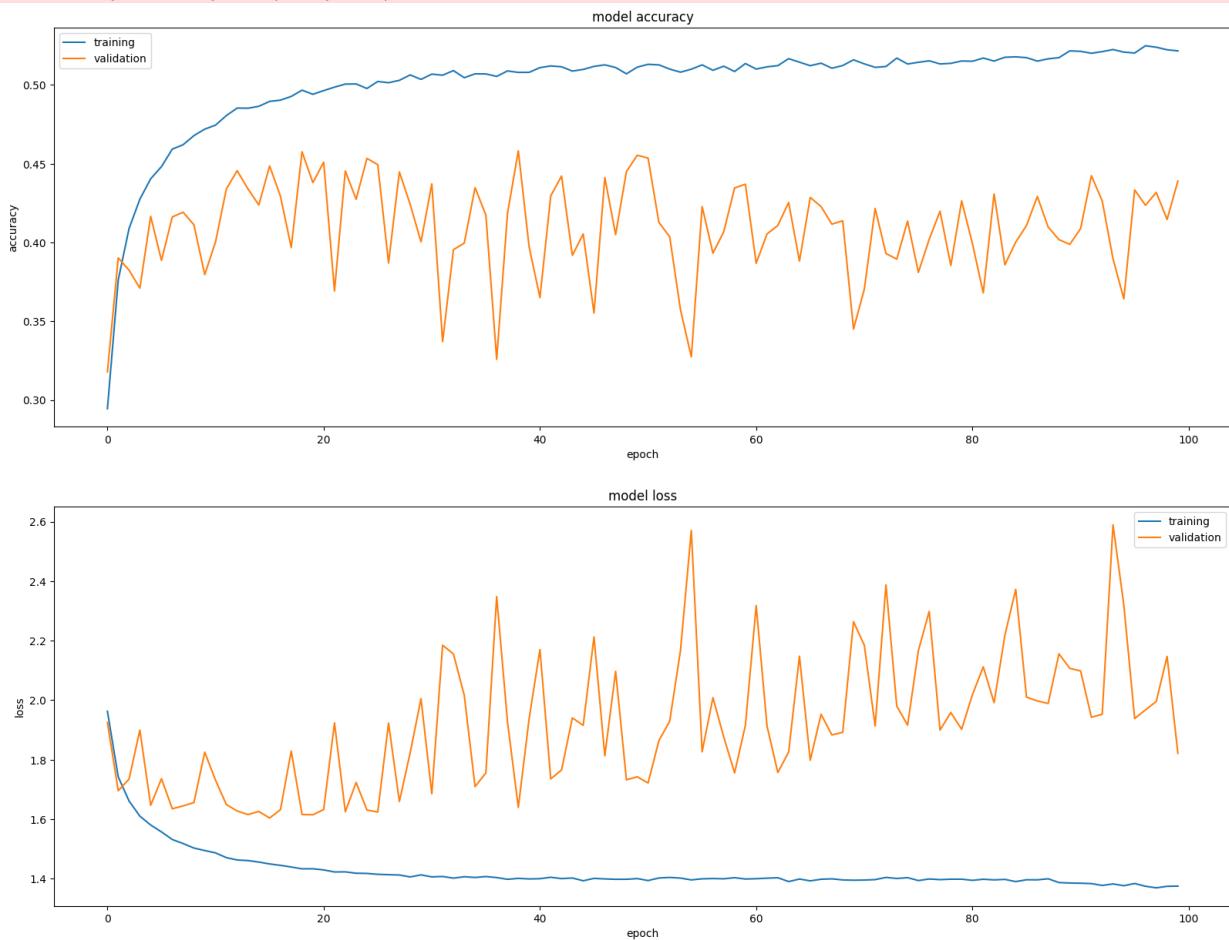
```
In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)
```

```
Out[ ]:    loss  accuracy  val_loss  val_accuracy
95   1.384      0.520     1.938      0.433
96   1.374      0.525     1.967      0.424
97   1.369      0.524     1.996      0.432
98   1.374      0.522     2.147      0.415
99   1.375      0.522     1.821      0.439
```

```
In [ ]: plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```

```
<ipython-input-8-353fbbe40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.
```

```
    ax = plt.subplot(subplot)
```



Let's examine precision and recall performance metrics for each of the prediction classes.

```
In [ ]: pred1= model.predict(x_test_reshaped)
pred1=np.argmax(pred1, axis=1)
```

```
313/313 [=====] - 1s 2ms/step
```

```
In [ ]: print_validation_report(y_test, pred1)
```

Classification Report

	precision	recall	f1-score	support
0	0.57	0.44	0.49	1000
1	0.64	0.53	0.58	1000
2	0.29	0.46	0.36	1000
3	0.34	0.32	0.33	1000
4	0.38	0.45	0.41	1000
5	0.41	0.33	0.36	1000
6	0.50	0.41	0.45	1000
7	0.52	0.52	0.52	1000
8	0.46	0.72	0.56	1000
9	0.66	0.33	0.44	1000
accuracy			0.45	10000
macro avg	0.48	0.45	0.45	10000
weighted avg	0.48	0.45	0.45	10000

Accuracy Score: 0.4507

Root Mean Square Error: 3.134852468617941

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

```
In [ ]: # Get the predicted classes:  
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning  
pred_classes = np.argmax(model.predict(x_test_reshaped), axis=-1)  
pred_classes;  
  
313/313 [=====] - 1s 2ms/step
```

Correlation matrix that measures the linear relationships

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html>

```
In [ ]: conf_mx = tf.math.confusion_matrix(y_test, pred_classes)  
conf_mx;
```

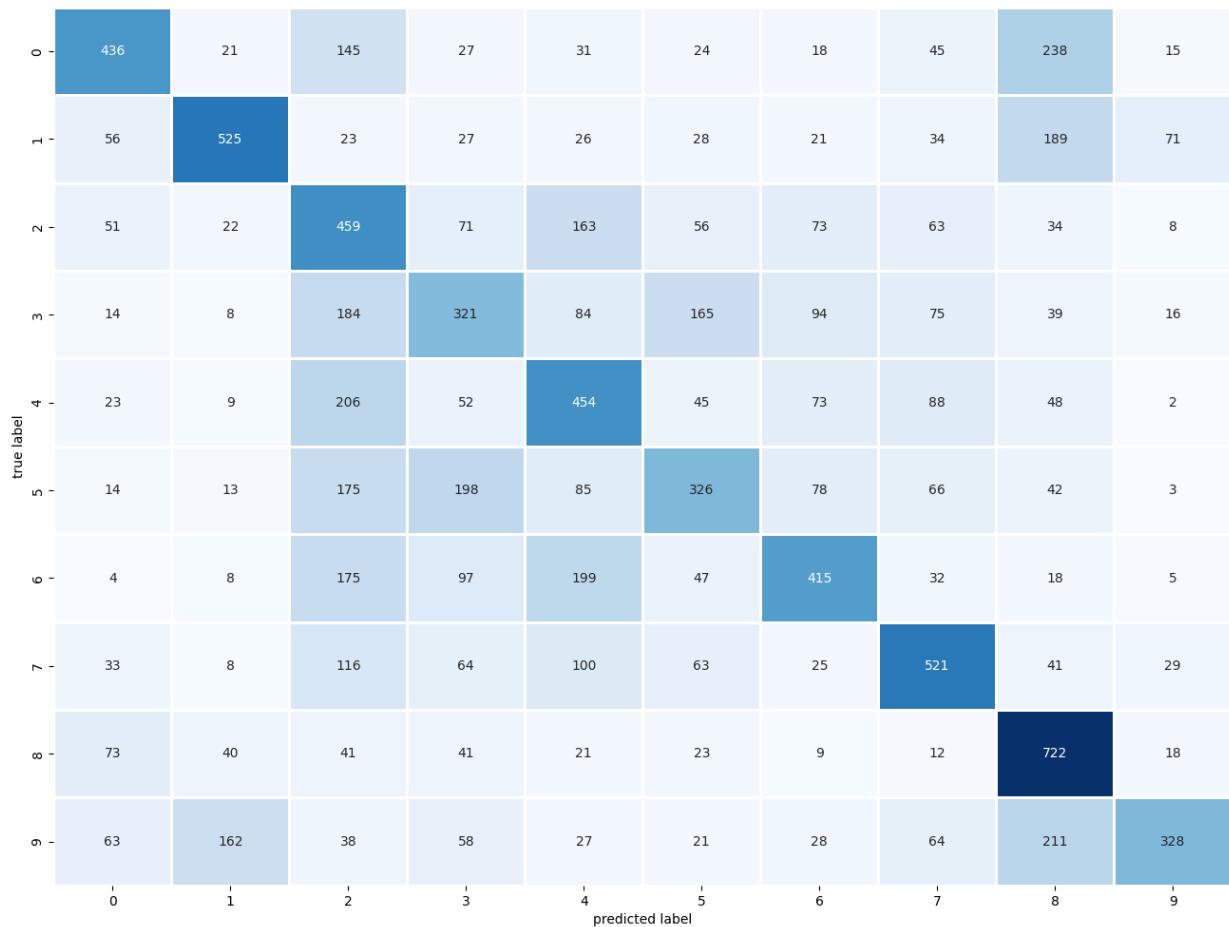
```
In [ ]: cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)  
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',  
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

Out[]:

	0	1	2	3	4	5	6	7	8	9
0	1.90%	0.95%	4.35%	54.40%	6.78%	20.13%	0.18%	4.79%	6.48%	0.06%
1	7.73%	11.46%	0.33%	0.22%	0.57%	0.03%	0.02%	0.13%	55.89%	23.62%
2	26.18%	1.80%	0.88%	0.42%	0.25%	0.16%	0.00%	0.17%	68.92%	1.24%
3	20.21%	6.06%	11.52%	2.03%	7.54%	1.33%	0.26%	1.88%	45.74%	3.43%
4	0.25%	0.01%	19.61%	3.92%	60.00%	3.42%	11.75%	0.78%	0.24%	0.01%
5	1.65%	0.33%	6.05%	19.09%	5.18%	13.00%	43.96%	10.00%	0.12%	0.62%
6	1.35%	23.88%	0.20%	10.66%	0.00%	6.81%	57.01%	0.06%	0.03%	0.01%
7	1.33%	2.53%	23.94%	16.22%	9.97%	12.23%	27.96%	4.01%	0.41%	1.39%
8	1.11%	0.01%	36.11%	18.80%	20.85%	18.78%	1.26%	2.67%	0.39%	0.01%
9	5.64%	56.95%	2.06%	1.58%	0.87%	0.66%	0.22%	0.86%	15.09%	16.07%
10	15.60%	0.13%	11.16%	10.03%	3.47%	8.08%	0.70%	0.14%	50.60%	0.09%
11	0.43%	17.26%	0.03%	0.11%	0.02%	0.05%	0.02%	0.30%	1.83%	79.94%
12	0.39%	1.07%	5.51%	28.33%	17.59%	24.78%	8.60%	8.57%	4.47%	0.70%
13	47.34%	0.03%	0.04%	0.00%	0.01%	0.00%	0.01%	52.56%	0.01%	0.00%
14	3.82%	37.96%	2.31%	0.48%	0.33%	0.48%	0.10%	1.59%	0.77%	52.15%
15	6.58%	0.24%	30.39%	7.21%	25.53%	6.72%	3.01%	0.81%	19.19%	0.32%
16	0.77%	0.62%	0.68%	38.65%	0.75%	48.49%	0.17%	9.42%	0.16%	0.30%
17	3.16%	0.05%	26.36%	22.39%	10.15%	27.59%	5.64%	3.83%	0.65%	0.17%
18	2.49%	0.11%	0.02%	0.01%	0.10%	0.00%	0.00%	0.01%	97.21%	0.05%
19	0.02%	0.01%	1.04%	4.81%	1.60%	3.58%	86.23%	2.65%	0.00%	0.07%

Let's create a confusion matrix that visualizes the model's performance on the testing data.

In []: `plot_confusion_matrix(y_test,pred_classes)`



3) Model 2 - Deep Neural Network with 3 Hidden Layers and No Regularization

3.1) Build The Model

```
In [ ]: k.clear_session()

model = Sequential([
    Dense(input_shape=[3072], units=256, activation = tf.nn.relu),
    Dense(units=128, activation = tf.nn.relu),
    Dense(units=64, activation = tf.nn.relu),
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

```
In [ ]: model.summary()
```

```
In [ ]: model.compile(optimizer='rmsprop',
                      loss = 'categorical_crossentropy',
                      metrics=['accuracy'])
```

```
In [ ]: #tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping
```

```
history = model.fit(  
    x_train_reshaped  
,y_train_encoded  
,epochs = 100  
,validation_data=(x_valid_reshaped, y_valid_encoded)  
,callbacks=[tf.keras.callbacks.ModelCheckpoint("Model_2", save_best_only=True,save_  
)
```

```
Epoch 1/100
1407/1407 [=====] - 10s 7ms/step - loss: 1.9275 - accuracy: 0.3027 - val_loss: 1.9308 - val_accuracy: 0.2916
Epoch 2/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.7281 - accuracy: 0.3806 - val_loss: 1.8621 - val_accuracy: 0.3290
Epoch 3/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.6509 - accuracy: 0.4103 - val_loss: 1.6934 - val_accuracy: 0.3912
Epoch 4/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.6015 - accuracy: 0.4306 - val_loss: 1.7719 - val_accuracy: 0.3700
Epoch 5/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.5711 - accuracy: 0.4394 - val_loss: 1.6946 - val_accuracy: 0.4026
Epoch 6/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.5468 - accuracy: 0.4503 - val_loss: 1.5937 - val_accuracy: 0.4270
Epoch 7/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.5336 - accuracy: 0.4582 - val_loss: 1.7954 - val_accuracy: 0.3850
Epoch 8/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.5164 - accuracy: 0.4629 - val_loss: 1.8313 - val_accuracy: 0.3994
Epoch 9/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.5058 - accuracy: 0.4688 - val_loss: 1.7800 - val_accuracy: 0.3968
Epoch 10/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4967 - accuracy: 0.4697 - val_loss: 1.7555 - val_accuracy: 0.4156
Epoch 11/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4959 - accuracy: 0.4717 - val_loss: 1.8330 - val_accuracy: 0.3838
Epoch 12/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4908 - accuracy: 0.4728 - val_loss: 1.6836 - val_accuracy: 0.4106
Epoch 13/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4851 - accuracy: 0.4748 - val_loss: 1.8728 - val_accuracy: 0.3860
Epoch 14/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4836 - accuracy: 0.4776 - val_loss: 1.6588 - val_accuracy: 0.4220
Epoch 15/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4784 - accuracy: 0.4801 - val_loss: 1.7129 - val_accuracy: 0.4180
Epoch 16/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.4703 - accuracy: 0.4804 - val_loss: 1.6725 - val_accuracy: 0.4352
Epoch 17/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4588 - accuracy: 0.4870 - val_loss: 1.8631 - val_accuracy: 0.3908
Epoch 18/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4547 - accuracy: 0.4887 - val_loss: 1.7270 - val_accuracy: 0.4222
Epoch 19/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4493 - accuracy: 0.4848 - val_loss: 1.6866 - val_accuracy: 0.4368
Epoch 20/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4494 - accuracy: 0.4924 - val_loss: 1.7033 - val_accuracy: 0.4326
```

```
Epoch 21/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4414 - accuracy: 0.4922 - val_loss: 1.8863 - val_accuracy: 0.4152
Epoch 22/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4353 - accuracy: 0.4942 - val_loss: 1.6261 - val_accuracy: 0.4534
Epoch 23/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4366 - accuracy: 0.4915 - val_loss: 1.8494 - val_accuracy: 0.4166
Epoch 24/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4275 - accuracy: 0.5011 - val_loss: 1.8183 - val_accuracy: 0.4096
Epoch 25/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4192 - accuracy: 0.5012 - val_loss: 1.6825 - val_accuracy: 0.4454
Epoch 26/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4163 - accuracy: 0.4995 - val_loss: 1.6884 - val_accuracy: 0.4342
Epoch 27/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4135 - accuracy: 0.5022 - val_loss: 1.7691 - val_accuracy: 0.4452
Epoch 28/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4146 - accuracy: 0.5015 - val_loss: 1.6871 - val_accuracy: 0.4274
Epoch 29/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4114 - accuracy: 0.5033 - val_loss: 1.6237 - val_accuracy: 0.4690
Epoch 30/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.4082 - accuracy: 0.5052 - val_loss: 1.8525 - val_accuracy: 0.4166
Epoch 31/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.4026 - accuracy: 0.5090 - val_loss: 1.8419 - val_accuracy: 0.4226
Epoch 32/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3997 - accuracy: 0.5105 - val_loss: 1.7533 - val_accuracy: 0.4434
Epoch 33/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3896 - accuracy: 0.5101 - val_loss: 1.9251 - val_accuracy: 0.4128
Epoch 34/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3903 - accuracy: 0.5118 - val_loss: 1.7500 - val_accuracy: 0.4576
Epoch 35/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3903 - accuracy: 0.5128 - val_loss: 1.8960 - val_accuracy: 0.3908
Epoch 36/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3855 - accuracy: 0.5157 - val_loss: 1.8679 - val_accuracy: 0.4354
Epoch 37/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3837 - accuracy: 0.5134 - val_loss: 1.7474 - val_accuracy: 0.4490
Epoch 38/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3796 - accuracy: 0.5185 - val_loss: 1.8252 - val_accuracy: 0.4168
Epoch 39/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3807 - accuracy: 0.5175 - val_loss: 1.7770 - val_accuracy: 0.4534
Epoch 40/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3783 - accuracy: 0.5171 - val_loss: 1.7727 - val_accuracy: 0.4496
```

```
Epoch 41/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3806 - accuracy: 0.5168 - val_loss: 1.8504 - val_accuracy: 0.4302
Epoch 42/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3711 - accuracy: 0.5186 - val_loss: 2.0938 - val_accuracy: 0.3954
Epoch 43/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3700 - accuracy: 0.5222 - val_loss: 1.8705 - val_accuracy: 0.4364
Epoch 44/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3698 - accuracy: 0.5221 - val_loss: 1.8949 - val_accuracy: 0.4476
Epoch 45/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3675 - accuracy: 0.5193 - val_loss: 1.8216 - val_accuracy: 0.4550
Epoch 46/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3547 - accuracy: 0.5260 - val_loss: 1.9662 - val_accuracy: 0.4424
Epoch 47/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3701 - accuracy: 0.5203 - val_loss: 1.9533 - val_accuracy: 0.4190
Epoch 48/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3676 - accuracy: 0.5204 - val_loss: 2.2311 - val_accuracy: 0.3532
Epoch 49/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3633 - accuracy: 0.5202 - val_loss: 1.9096 - val_accuracy: 0.4338
Epoch 50/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3560 - accuracy: 0.5249 - val_loss: 1.8713 - val_accuracy: 0.4110
Epoch 51/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3570 - accuracy: 0.5272 - val_loss: 1.8791 - val_accuracy: 0.4478
Epoch 52/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3634 - accuracy: 0.5226 - val_loss: 1.9698 - val_accuracy: 0.4304
Epoch 53/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3585 - accuracy: 0.5262 - val_loss: 1.8437 - val_accuracy: 0.4394
Epoch 54/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3527 - accuracy: 0.5284 - val_loss: 1.8854 - val_accuracy: 0.4540
Epoch 55/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3601 - accuracy: 0.5247 - val_loss: 1.9245 - val_accuracy: 0.4472
Epoch 56/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3586 - accuracy: 0.5245 - val_loss: 2.6170 - val_accuracy: 0.3638
Epoch 57/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3736 - accuracy: 0.5235 - val_loss: 2.2633 - val_accuracy: 0.4142
Epoch 58/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3599 - accuracy: 0.5268 - val_loss: 1.9610 - val_accuracy: 0.4380
Epoch 59/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3656 - accuracy: 0.5254 - val_loss: 1.8511 - val_accuracy: 0.4478
Epoch 60/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3609 - accuracy: 0.5259 - val_loss: 2.3319 - val_accuracy: 0.4100
```

```
Epoch 61/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3629 - accuracy: 0.5290 - val_loss: 1.9402 - val_accuracy: 0.4298
Epoch 62/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3652 - accuracy: 0.5235 - val_loss: 2.1160 - val_accuracy: 0.4324
Epoch 63/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3634 - accuracy: 0.5261 - val_loss: 1.8739 - val_accuracy: 0.4080
Epoch 64/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3602 - accuracy: 0.5274 - val_loss: 2.0046 - val_accuracy: 0.4172
Epoch 65/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3613 - accuracy: 0.5255 - val_loss: 1.9831 - val_accuracy: 0.4398
Epoch 66/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3525 - accuracy: 0.5283 - val_loss: 2.0516 - val_accuracy: 0.4274
Epoch 67/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3575 - accuracy: 0.5240 - val_loss: 2.4164 - val_accuracy: 0.3844
Epoch 68/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3492 - accuracy: 0.5291 - val_loss: 2.1082 - val_accuracy: 0.4274
Epoch 69/100
1407/1407 [=====] - 11s 8ms/step - loss: 1.3528 - accuracy: 0.5282 - val_loss: 1.9232 - val_accuracy: 0.4524
Epoch 70/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3510 - accuracy: 0.5308 - val_loss: 2.1282 - val_accuracy: 0.4106
Epoch 71/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3419 - accuracy: 0.5297 - val_loss: 2.1329 - val_accuracy: 0.4378
Epoch 72/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3384 - accuracy: 0.5298 - val_loss: 2.1621 - val_accuracy: 0.3966
Epoch 73/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3408 - accuracy: 0.5314 - val_loss: 2.0329 - val_accuracy: 0.4310
Epoch 74/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3396 - accuracy: 0.5316 - val_loss: 2.8919 - val_accuracy: 0.3742
Epoch 75/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3319 - accuracy: 0.5342 - val_loss: 2.0771 - val_accuracy: 0.4034
Epoch 76/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3331 - accuracy: 0.5356 - val_loss: 2.0238 - val_accuracy: 0.4534
Epoch 77/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3356 - accuracy: 0.5363 - val_loss: 2.2890 - val_accuracy: 0.4118
Epoch 78/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3417 - accuracy: 0.5344 - val_loss: 2.4500 - val_accuracy: 0.3908
Epoch 79/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3404 - accuracy: 0.5323 - val_loss: 1.9422 - val_accuracy: 0.4330
Epoch 80/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3408 - accuracy: 0.5349 - val_loss: 2.0116 - val_accuracy: 0.4494
```

```
Epoch 81/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3436 - accuracy: 0.5342 - val_loss: 2.3998 - val_accuracy: 0.4096
Epoch 82/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3383 - accuracy: 0.5316 - val_loss: 2.2872 - val_accuracy: 0.4252
Epoch 83/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3361 - accuracy: 0.5341 - val_loss: 2.2138 - val_accuracy: 0.4174
Epoch 84/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3467 - accuracy: 0.5290 - val_loss: 2.0992 - val_accuracy: 0.4194
Epoch 85/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3503 - accuracy: 0.5307 - val_loss: 2.0373 - val_accuracy: 0.4226
Epoch 86/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3506 - accuracy: 0.5297 - val_loss: 2.2714 - val_accuracy: 0.3780
Epoch 87/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3501 - accuracy: 0.5281 - val_loss: 2.2001 - val_accuracy: 0.3932
Epoch 88/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3490 - accuracy: 0.5318 - val_loss: 2.6532 - val_accuracy: 0.3500
Epoch 89/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3447 - accuracy: 0.5300 - val_loss: 2.1732 - val_accuracy: 0.4310
Epoch 90/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3532 - accuracy: 0.5311 - val_loss: 1.9745 - val_accuracy: 0.4538
Epoch 91/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3332 - accuracy: 0.5356 - val_loss: 2.2391 - val_accuracy: 0.4136
Epoch 92/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3409 - accuracy: 0.5315 - val_loss: 2.1020 - val_accuracy: 0.4136
Epoch 93/100
1407/1407 [=====] - 9s 7ms/step - loss: 1.3435 - accuracy: 0.5327 - val_loss: 2.4190 - val_accuracy: 0.4360
Epoch 94/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3449 - accuracy: 0.5335 - val_loss: 2.2376 - val_accuracy: 0.4110
Epoch 95/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3466 - accuracy: 0.5300 - val_loss: 2.2554 - val_accuracy: 0.3994
Epoch 96/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3391 - accuracy: 0.5340 - val_loss: 2.1118 - val_accuracy: 0.4254
Epoch 97/100
1407/1407 [=====] - 8s 6ms/step - loss: 1.3279 - accuracy: 0.5365 - val_loss: 2.0475 - val_accuracy: 0.4232
Epoch 98/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3332 - accuracy: 0.5349 - val_loss: 2.7488 - val_accuracy: 0.4024
Epoch 99/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3375 - accuracy: 0.5346 - val_loss: 2.1375 - val_accuracy: 0.4438
Epoch 100/100
1407/1407 [=====] - 9s 6ms/step - loss: 1.3233 - accuracy: 0.5372 - val_loss: 2.1652 - val_accuracy: 0.4538
```

3.2) Evaluate Model Performance

Now that we've fit our model, let's apply the model to the test dataset and subsequently evaluate its performance.

```
In [ ]: model = tf.keras.models.load_model("Model_2")
print(f"Training accuracy: {model.evaluate(x_train_reshaped, y_train_encoded)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_reshaped, y_valid_encoded)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_reshaped, y_test_encoded)[1]:.3f}")

1407/1407 [=====] - 3s 2ms/step - loss: 1.4993 - accuracy: 0.4570
Training accuracy: 0.457
157/157 [=====] - 0s 3ms/step - loss: 1.5937 - accuracy: 0.4270
Validation accuracy: 0.427
313/313 [=====] - 1s 2ms/step - loss: 1.5677 - accuracy: 0.4307
Test accuracy: 0.431
```

```
In [ ]: # loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

```
In [ ]: preds = model.predict(x_test_reshaped)
print('shape of preds: ', preds.shape)

313/313 [=====] - 1s 2ms/step
shape of preds: (10000, 10)
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

```
In [ ]: plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color=col)
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

```
In [ ]: history_dict = history.history
history_dict.keys()
```

```
Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

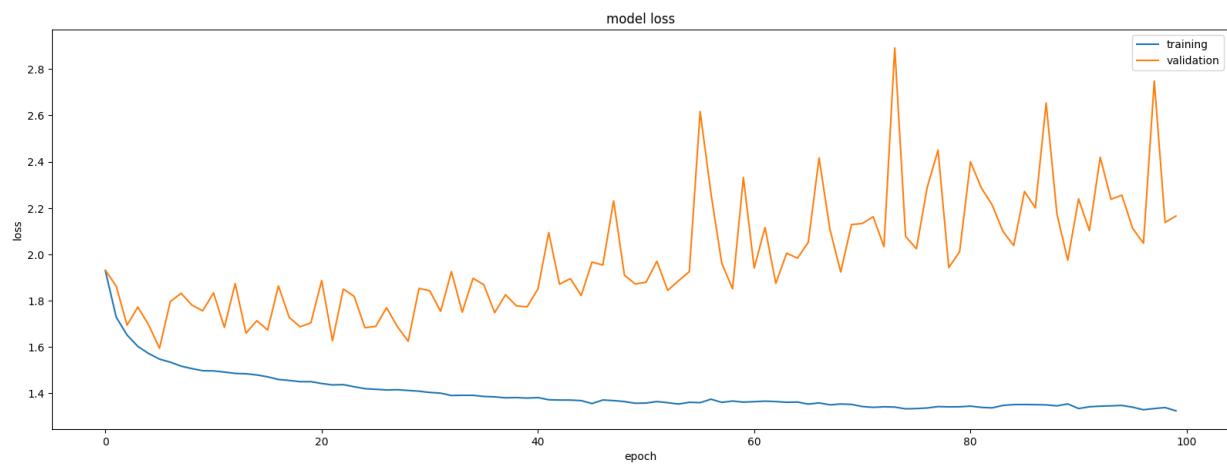
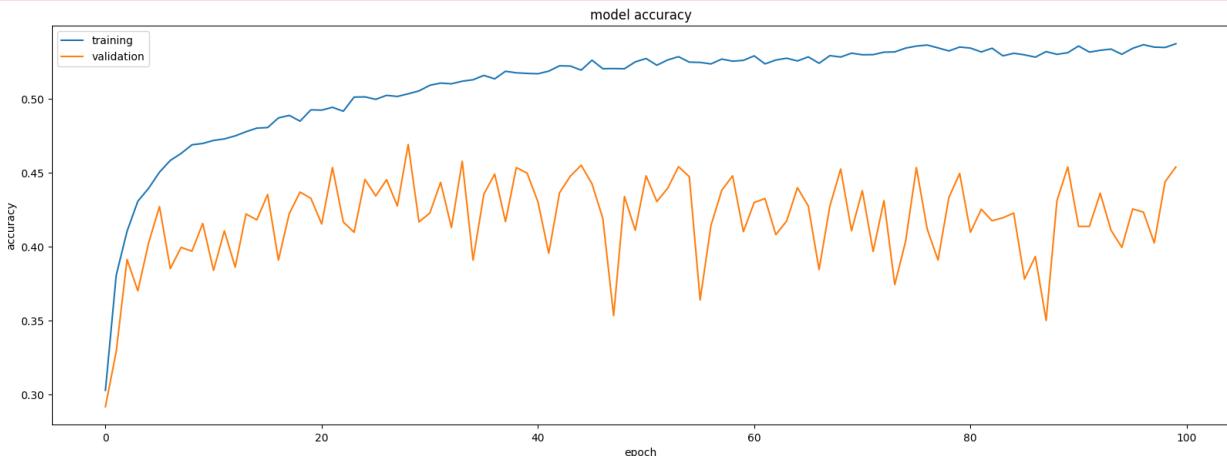
```
In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)
```

```
Out[ ]:    loss  accuracy  val_loss  val_accuracy
95   1.339      0.534     2.112      0.425
96   1.328      0.537     2.048      0.423
97   1.333      0.535     2.749      0.402
98   1.338      0.535     2.138      0.444
99   1.323      0.537     2.165      0.454
```

```
In [ ]: plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```

```
<ipython-input-8-353fbbe40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.
```

```
    ax = plt.subplot(subplot)
```



Let's examine precision and recall performance metrics for each of the prediction classes.

```
In [ ]: pred1= model.predict(x_test_reshaped)
pred1=np.argmax(pred1, axis=1)
```

```
313/313 [=====] - 1s 2ms/step
```

```
In [ ]: print_validation_report(y_test, pred1)
```

Classification Report

	precision	recall	f1-score	support
0	0.55	0.42	0.47	1000
1	0.70	0.29	0.41	1000
2	0.28	0.42	0.34	1000
3	0.34	0.24	0.28	1000
4	0.48	0.18	0.26	1000
5	0.35	0.44	0.39	1000
6	0.44	0.56	0.49	1000
7	0.61	0.42	0.50	1000
8	0.53	0.61	0.56	1000
9	0.39	0.73	0.51	1000
accuracy			0.43	10000
macro avg	0.47	0.43	0.42	10000
weighted avg	0.47	0.43	0.42	10000

Accuracy Score: 0.4307

Root Mean Square Error: 3.391386147285502

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

```
In [ ]: # Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_reshaped), axis=-1)
pred_classes;
```

313/313 [=====] - 1s 3ms/step

Correlation matrix that measures the linear relationships

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html>

```
In [ ]: conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;
```

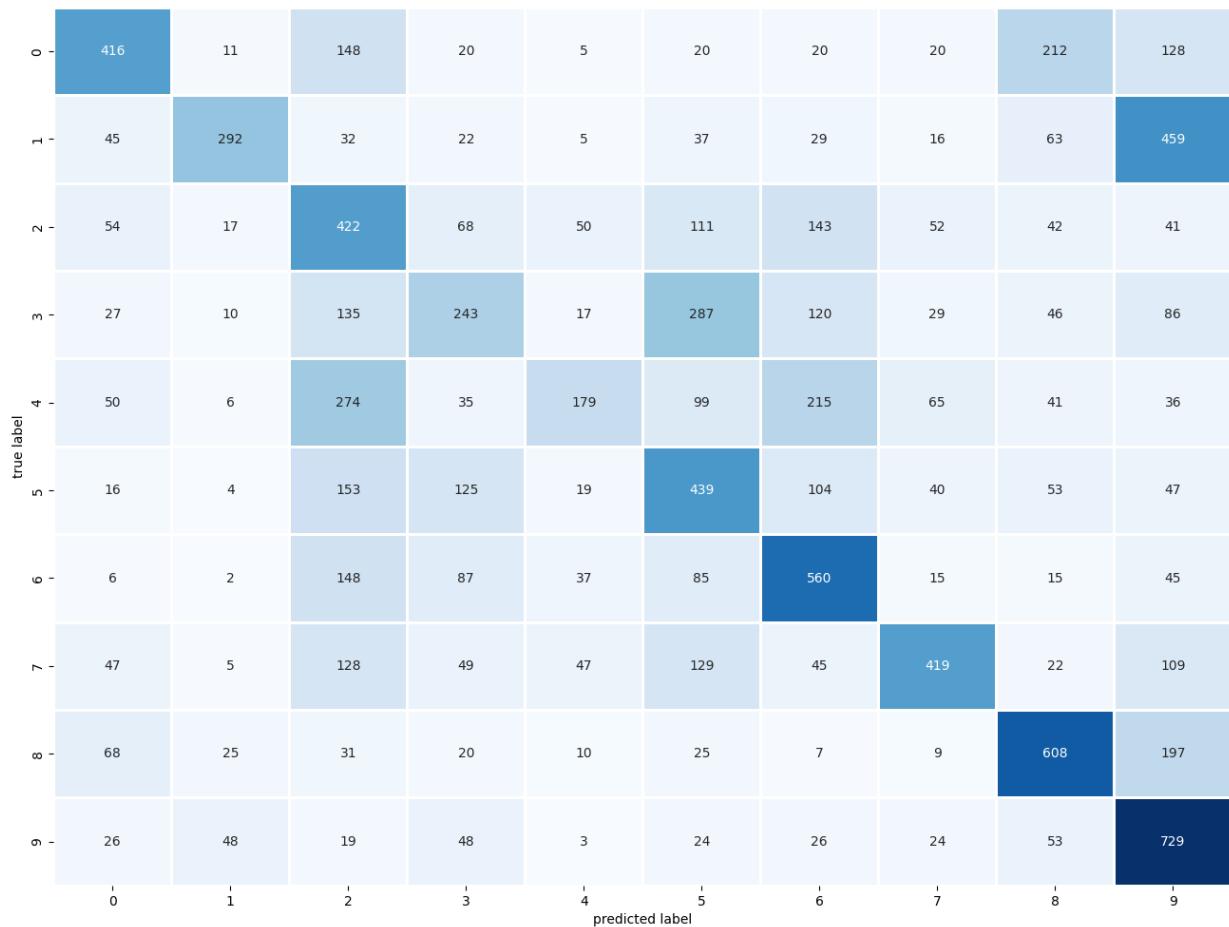
```
In [ ]: cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

Out[]:

	0	1	2	3	4	5	6	7	8	9
0	7.41%	4.14%	12.04%	22.17%	6.06%	18.56%	17.30%	1.78%	8.08%	2.46%
1	0.33%	5.26%	0.04%	0.09%	0.00%	0.02%	0.00%	0.04%	2.85%	91.38%
2	11.65%	9.80%	0.25%	0.19%	0.05%	0.15%	0.00%	0.20%	33.73%	43.98%
3	21.69%	17.01%	9.76%	2.76%	2.69%	2.42%	0.45%	2.86%	18.16%	22.21%
4	0.54%	0.10%	19.55%	6.24%	30.00%	10.74%	30.94%	1.40%	0.41%	0.07%
5	1.56%	1.86%	13.02%	13.54%	9.92%	11.04%	40.88%	3.11%	0.31%	4.76%
6	5.92%	6.43%	6.03%	30.02%	0.21%	37.09%	11.05%	0.92%	0.81%	1.51%
7	1.54%	0.40%	29.28%	8.39%	26.68%	7.09%	21.59%	3.72%	0.75%	0.56%
8	4.96%	0.41%	31.06%	22.41%	9.21%	20.54%	3.32%	5.42%	1.55%	1.11%
9	1.47%	33.28%	2.81%	5.05%	0.12%	2.23%	0.49%	1.01%	7.39%	46.14%
10	11.58%	0.62%	4.50%	4.16%	2.10%	7.10%	0.70%	0.89%	67.19%	1.17%
11	0.15%	10.02%	0.09%	0.27%	0.01%	0.08%	0.01%	0.05%	3.85%	85.47%
12	3.44%	3.15%	14.42%	14.93%	9.95%	22.05%	21.83%	4.13%	3.41%	2.68%
13	32.48%	0.44%	3.24%	3.28%	0.38%	1.70%	1.11%	53.73%	3.05%	0.60%
14	0.79%	45.88%	0.43%	0.08%	0.00%	0.05%	0.00%	0.26%	0.25%	52.24%
15	5.57%	2.30%	2.65%	5.60%	2.01%	10.08%	0.92%	1.22%	66.54%	3.12%
16	0.34%	0.66%	15.41%	12.09%	0.30%	62.31%	0.64%	1.24%	0.18%	6.83%
17	6.10%	0.69%	14.47%	22.33%	23.11%	14.25%	4.56%	8.14%	2.30%	4.05%
18	2.30%	2.04%	0.04%	0.32%	0.02%	0.02%	0.01%	0.40%	78.08%	16.77%
19	0.17%	0.24%	3.14%	10.34%	1.42%	10.96%	70.26%	0.86%	0.05%	2.57%

Let's create a confusion matrix that visualizes the model's performance on the testing data.

In []: `plot_confusion_matrix(y_test,pred_classes)`



4) Model 3 - CNN with 2 Convolutional / Max Pooling Layers and No Regularization

4.1) Build The Model

We use a Sequential class defined in Keras to create our model.

```
In [ ]: k.clear_session()
model = Sequential([
    Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_shape=(28, 28, 1)),
    MaxPool2D((2, 2), strides=2),
    Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu),
    MaxPool2D((2, 2), strides=2),
    Flatten(),
    Dense(units=384, activation=tf.nn.softmax),
    Dense(units=10, activation=tf.nn.softmax)
])
```

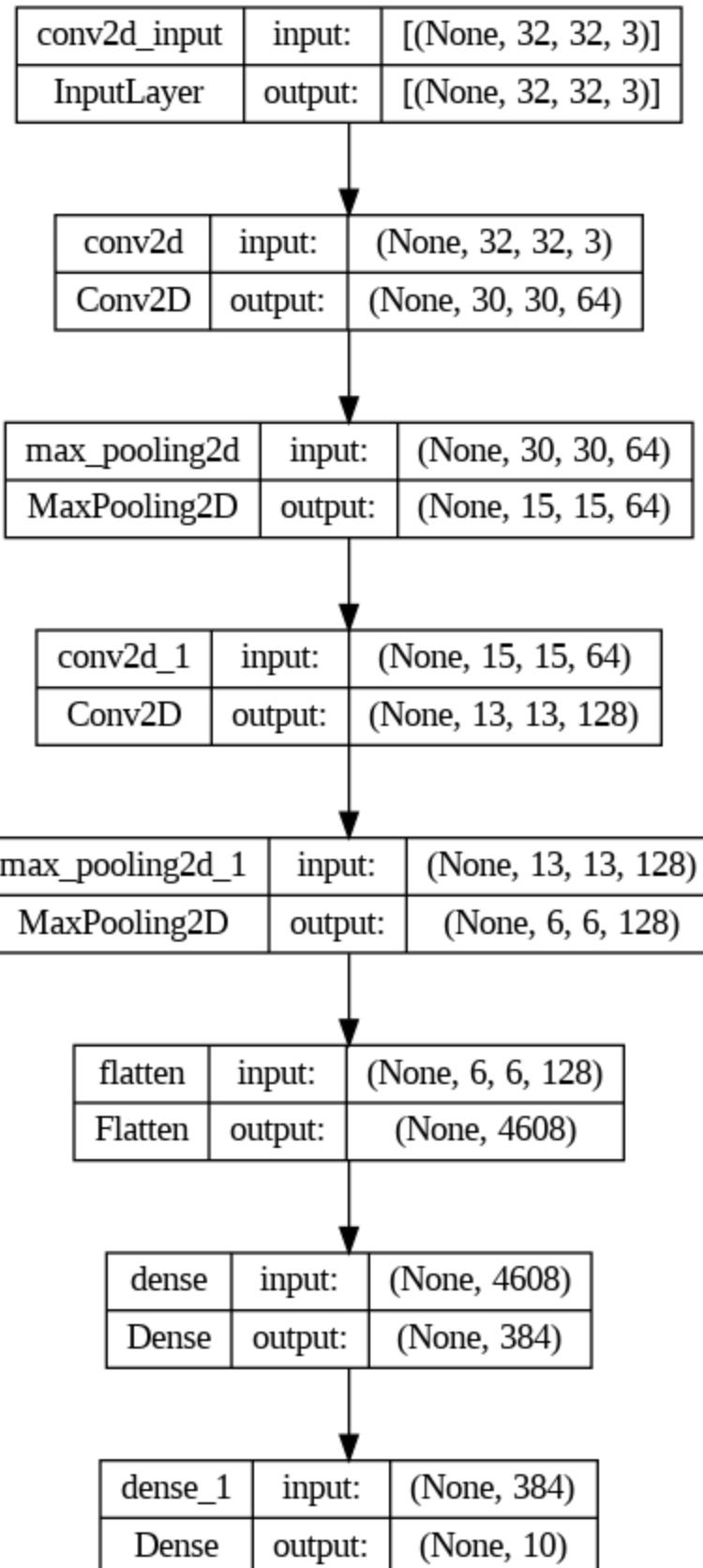
```
In [ ]: model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 64)	1792
max_pooling2d (MaxPooling2D)	(None, 15, 15, 64)	0
conv2d_1 (Conv2D)	(None, 13, 13, 128)	73856
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 128)	0
flatten (Flatten)	(None, 4608)	0
dense (Dense)	(None, 384)	1769856
dense_1 (Dense)	(None, 10)	3850
<hr/>		
Total params: 1849354 (7.05 MB)		
Trainable params: 1849354 (7.05 MB)		
Non-trainable params: 0 (0.00 Byte)		

In []: `tf.keras.utils.plot_model(model, "CIFAR10.png", show_shapes=True)`

Out[]:



Let's now compile and train the model.

tf.keras.losses.SparseCategoricalCrossentropy

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Module: tf.keras.callbacks**tf.keras.callbacks.EarlyStopping**https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping**tf.keras.callbacks.ModelCheckpoint**https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

```
In [ ]: model.compile(optimizer='adam',
                     loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
                     metrics=['accuracy'])
```

```
In [ ]: history = model.fit(x_train_norm
                           ,y_train_split
                           ,epochs=30
                           ,validation_data=(x_valid_norm, y_valid_split)
                           ,callbacks=[
                           tf.keras.callbacks.ModelCheckpoint("Model_3", save_best_only=True,
                           )
                           ])
```

```
Epoch 1/30
1407/1407 [=====] - 98s 70ms/step - loss: 1.3198 - accuracy: 0.4690 - val_loss: 1.4333 - val_accuracy: 0.4354
Epoch 2/30
1407/1407 [=====] - 96s 69ms/step - loss: 1.2845 - accuracy: 0.4818 - val_loss: 1.3976 - val_accuracy: 0.4440
Epoch 3/30
1407/1407 [=====] - 95s 67ms/step - loss: 1.2508 - accuracy: 0.4999 - val_loss: 1.3994 - val_accuracy: 0.4358
Epoch 4/30
1407/1407 [=====] - 94s 67ms/step - loss: 1.2232 - accuracy: 0.5100 - val_loss: 1.4112 - val_accuracy: 0.4492
Epoch 5/30
1407/1407 [=====] - 96s 68ms/step - loss: 1.1986 - accuracy: 0.5194 - val_loss: 1.3841 - val_accuracy: 0.4584
Epoch 6/30
1407/1407 [=====] - 94s 67ms/step - loss: 1.1724 - accuracy: 0.5311 - val_loss: 1.3909 - val_accuracy: 0.4626
Epoch 7/30
1407/1407 [=====] - 96s 68ms/step - loss: 1.1507 - accuracy: 0.5368 - val_loss: 1.3791 - val_accuracy: 0.4676
Epoch 8/30
1407/1407 [=====] - 94s 67ms/step - loss: 1.1289 - accuracy: 0.5459 - val_loss: 1.3519 - val_accuracy: 0.4848
Epoch 9/30
1407/1407 [=====] - 94s 67ms/step - loss: 1.1003 - accuracy: 0.5671 - val_loss: 1.3680 - val_accuracy: 0.4920
Epoch 10/30
1407/1407 [=====] - 168s 119ms/step - loss: 1.0725 - accuracy: 0.5812 - val_loss: 1.3475 - val_accuracy: 0.4980
Epoch 11/30
1407/1407 [=====] - 102s 73ms/step - loss: 1.0434 - accuracy: 0.5961 - val_loss: 1.3357 - val_accuracy: 0.4998
Epoch 12/30
1407/1407 [=====] - 109s 77ms/step - loss: 1.0084 - accuracy: 0.6177 - val_loss: 1.3155 - val_accuracy: 0.5286
Epoch 13/30
1407/1407 [=====] - 95s 67ms/step - loss: 0.9565 - accuracy: 0.6390 - val_loss: 1.3048 - val_accuracy: 0.5342
Epoch 14/30
1407/1407 [=====] - 95s 67ms/step - loss: 0.9204 - accuracy: 0.6547 - val_loss: 1.2923 - val_accuracy: 0.5510
Epoch 15/30
1407/1407 [=====] - 92s 66ms/step - loss: 0.8891 - accuracy: 0.6689 - val_loss: 1.3302 - val_accuracy: 0.5346
Epoch 16/30
1407/1407 [=====] - 92s 65ms/step - loss: 0.8651 - accuracy: 0.6730 - val_loss: 1.2828 - val_accuracy: 0.5524
Epoch 17/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.8392 - accuracy: 0.6866 - val_loss: 1.2947 - val_accuracy: 0.5570
Epoch 18/30
1407/1407 [=====] - 93s 66ms/step - loss: 0.8082 - accuracy: 0.6995 - val_loss: 1.2930 - val_accuracy: 0.5516
Epoch 19/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.7891 - accuracy: 0.7042 - val_loss: 1.3256 - val_accuracy: 0.5540
Epoch 20/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.7682 - accuracy: 0.7172 - val_loss: 1.3367 - val_accuracy: 0.5562
```

```

Epoch 21/30
1407/1407 [=====] - 97s 69ms/step - loss: 0.7475 - accuracy: 0.7245 - val_loss: 1.3216 - val_accuracy: 0.5608
Epoch 22/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.7291 - accuracy: 0.7297 - val_loss: 1.3980 - val_accuracy: 0.5468
Epoch 23/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.7069 - accuracy: 0.7408 - val_loss: 1.3441 - val_accuracy: 0.5650
Epoch 24/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.6838 - accuracy: 0.7527 - val_loss: 1.3727 - val_accuracy: 0.5688
Epoch 25/30
1407/1407 [=====] - 95s 67ms/step - loss: 0.6575 - accuracy: 0.7686 - val_loss: 1.3635 - val_accuracy: 0.5776
Epoch 26/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.6248 - accuracy: 0.7828 - val_loss: 1.3408 - val_accuracy: 0.5912
Epoch 27/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.6002 - accuracy: 0.7924 - val_loss: 1.3553 - val_accuracy: 0.5940
Epoch 28/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.5680 - accuracy: 0.8083 - val_loss: 1.3462 - val_accuracy: 0.5972
Epoch 29/30
1407/1407 [=====] - 93s 66ms/step - loss: 0.5460 - accuracy: 0.8151 - val_loss: 1.3553 - val_accuracy: 0.6096
Epoch 30/30
1407/1407 [=====] - 94s 67ms/step - loss: 0.5213 - accuracy: 0.8241 - val_loss: 1.3839 - val_accuracy: 0.6076

```

4.2) Evaluate Model Performance

```

In [ ]: model = tf.keras.models.load_model("Model_3")
print(f"Training accuracy: {model.evaluate(x_train_norm, y_train_split)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_norm, y_valid_split)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_norm, y_test)[1]:.3f}")

1407/1407 [=====] - 27s 19ms/step - loss: 0.7945 - accuracy: 0.7074
Training accuracy: 0.707
157/157 [=====] - 3s 17ms/step - loss: 1.2828 - accuracy: 0.5524
Validation accuracy: 0.552
313/313 [=====] - 6s 20ms/step - loss: 1.3131 - accuracy: 0.5449
Test accuracy: 0.545

```

```

In [ ]: preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)

313/313 [=====] - 9s 28ms/step
shape of preds: (10000, 10)

```

```

In [ ]: history_dict = history.history
history_dict.keys()

Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])

```

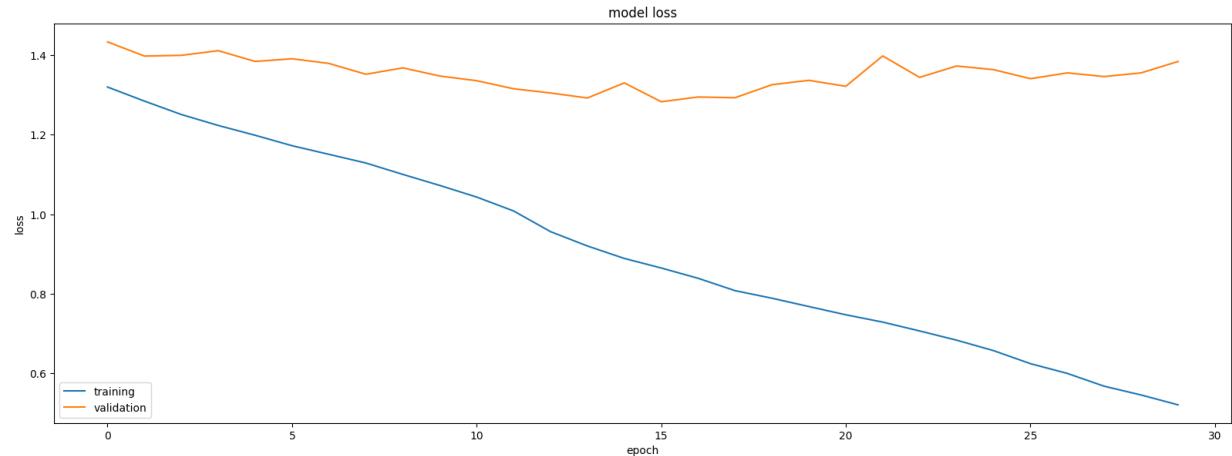
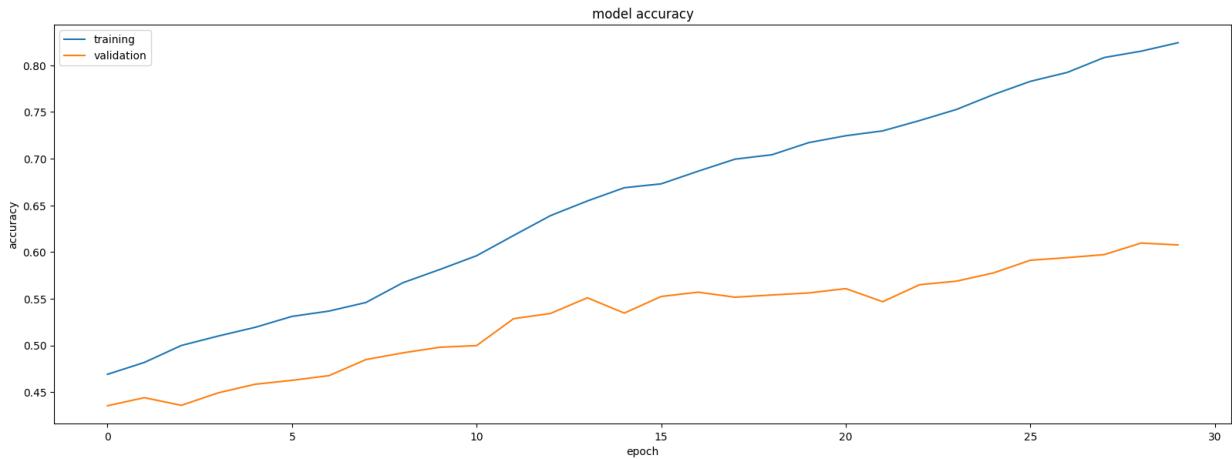
```
In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)
```

```
Out[ ]:   loss  accuracy  val_loss  val_accuracy
25    0.625      0.783     1.341      0.591
26    0.600      0.792     1.355      0.594
27    0.568      0.808     1.346      0.597
28    0.546      0.815     1.355      0.610
29    0.521      0.824     1.384      0.608
```

```
In [ ]: plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```

<ipython-input-8-353fb4e40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.

```
ax = plt.subplot(subplot)
```



Let's examine the precision, recall, F1-score, and confusion matrix.

```
In [ ]: pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)
```

313/313 [=====] - 6s 18ms/step

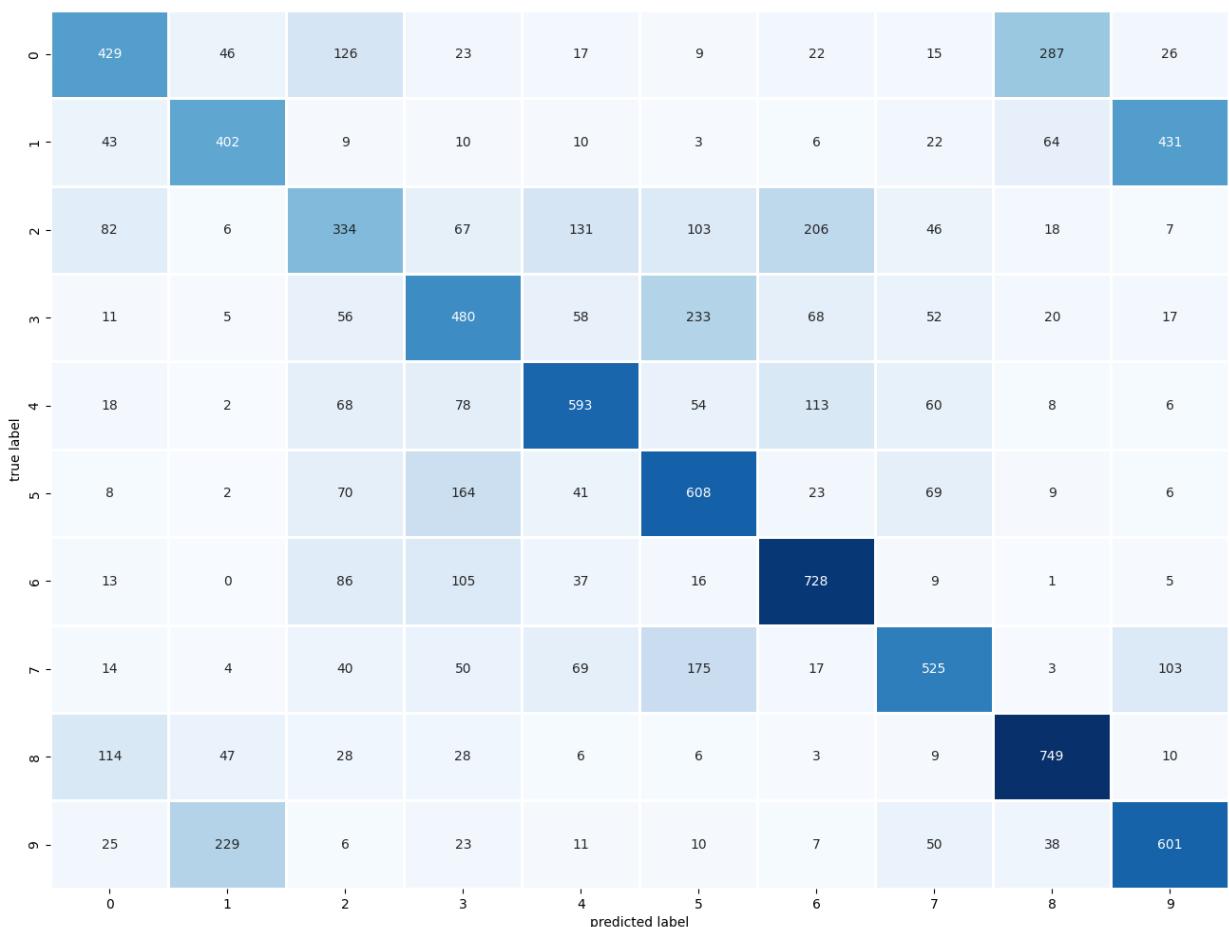
In []: `print_validation_report(y_test, pred1)`

Classification Report

	precision	recall	f1-score	support
0	0.57	0.43	0.49	1000
1	0.54	0.40	0.46	1000
2	0.41	0.33	0.37	1000
3	0.47	0.48	0.47	1000
4	0.61	0.59	0.60	1000
5	0.50	0.61	0.55	1000
6	0.61	0.73	0.66	1000
7	0.61	0.53	0.57	1000
8	0.63	0.75	0.68	1000
9	0.50	0.60	0.54	1000
accuracy			0.54	10000
macro avg	0.54	0.54	0.54	10000
weighted avg	0.54	0.54	0.54	10000

Accuracy Score: 0.5449

Root Mean Square Error: 3.2801371922527873

In []: `plot_confusion_matrix(y_test, pred1)`

Load HDF5 Model Format

tf.keras.models.load_model

https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model

```
In [ ]: model = tf.keras.models.load_model('Model_3')
```

```
In [ ]: preds = model.predict(x_test_norm)
```

313/313 [=====] - 5s 17ms/step

```
In [ ]: preds.shape
```

```
Out[ ]: (10000, 10)
```

Let's examine the predictions

```
In [ ]: cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
```

```
In [ ]: df = pd.DataFrame(preds[0:20], columns = ['airplane'
                                                 , 'automobile'
                                                 , 'bird'
                                                 , 'cat'
                                                 , 'deer'
                                                 , 'dog'
                                                 , 'frog'
                                                 , 'horse'
                                                 , 'ship'
                                                 , 'truck'])
```

```
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

Out[]:

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0	0.18%	0.01%	1.11%	85.68%	1.43%	7.94%	0.96%	1.58%	0.96%	0.15%
1	24.21%	0.69%	0.54%	0.01%	0.01%	0.00%	0.00%	0.00%	74.47%	0.07%
2	31.88%	2.26%	1.10%	0.06%	0.05%	0.01%	0.00%	0.00%	64.27%	0.37%
3	47.68%	1.06%	2.41%	0.08%	0.06%	0.01%	0.00%	0.00%	48.54%	0.16%
4	1.11%	0.10%	5.93%	3.97%	76.99%	1.05%	9.00%	1.68%	0.05%	0.11%
5	0.81%	0.01%	19.14%	1.40%	10.06%	0.43%	67.20%	0.92%	0.00%	0.02%
6	0.04%	38.56%	0.01%	0.00%	0.06%	0.00%	0.00%	9.42%	0.01%	51.89%
7	0.49%	0.00%	17.83%	0.56%	5.93%	0.18%	74.85%	0.17%	0.00%	0.00%
8	0.21%	0.19%	0.09%	89.67%	0.86%	0.47%	4.54%	0.16%	2.90%	0.91%
9	0.39%	57.52%	0.03%	0.01%	0.09%	0.00%	0.00%	1.61%	0.11%	40.24%
10	13.98%	0.54%	31.31%	6.23%	39.92%	2.72%	1.56%	1.15%	2.08%	0.51%
11	0.03%	45.70%	0.00%	0.00%	0.03%	0.00%	0.00%	5.47%	0.00%	48.77%
12	0.01%	0.00%	3.27%	13.67%	0.37%	74.11%	0.01%	8.53%	0.03%	0.00%
13	0.68%	39.79%	0.14%	0.04%	0.60%	0.01%	0.02%	10.90%	0.15%	47.65%
14	0.02%	45.34%	0.00%	0.00%	0.03%	0.00%	0.00%	5.65%	0.00%	48.95%
15	5.02%	1.80%	1.28%	64.69%	3.63%	0.67%	8.60%	0.15%	11.58%	2.58%
16	0.01%	0.00%	3.49%	12.91%	0.42%	67.52%	0.01%	15.60%	0.03%	0.00%
17	0.72%	19.46%	0.35%	0.23%	1.64%	0.07%	0.10%	33.86%	0.17%	43.40%
18	26.11%	1.08%	0.64%	0.02%	0.02%	0.00%	0.00%	0.00%	72.01%	0.13%
19	0.25%	0.00%	10.58%	0.19%	4.28%	0.06%	84.56%	0.07%	0.00%	0.00%

In []:

```
(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[98]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
,'automobile'
,'bird'
,'cat'
,'deer'
,'dog'
,'frog'
,'horse'
,'ship'
,'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))

```

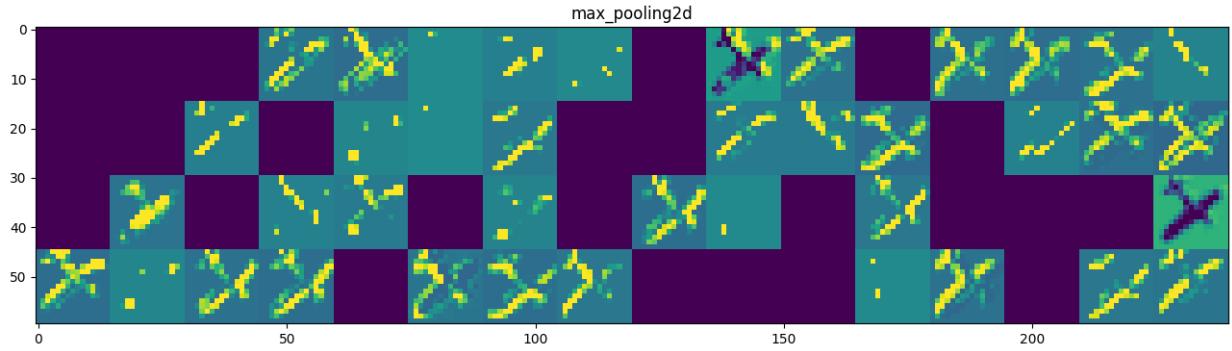
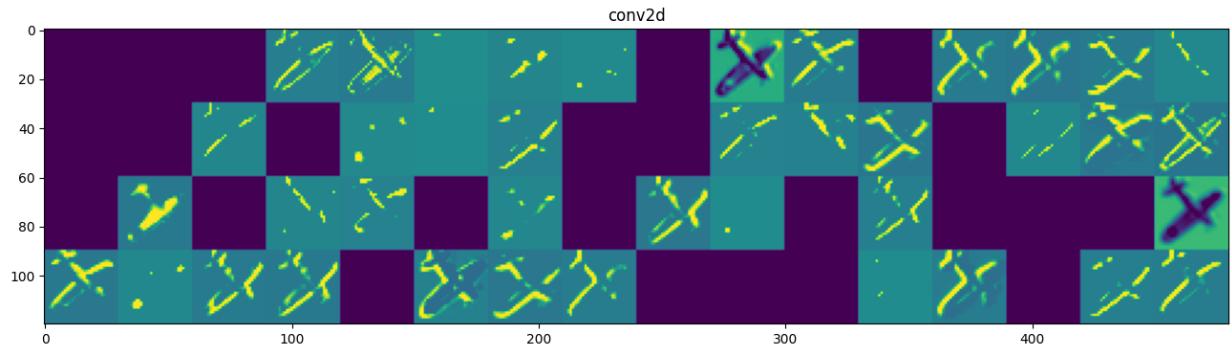
```
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

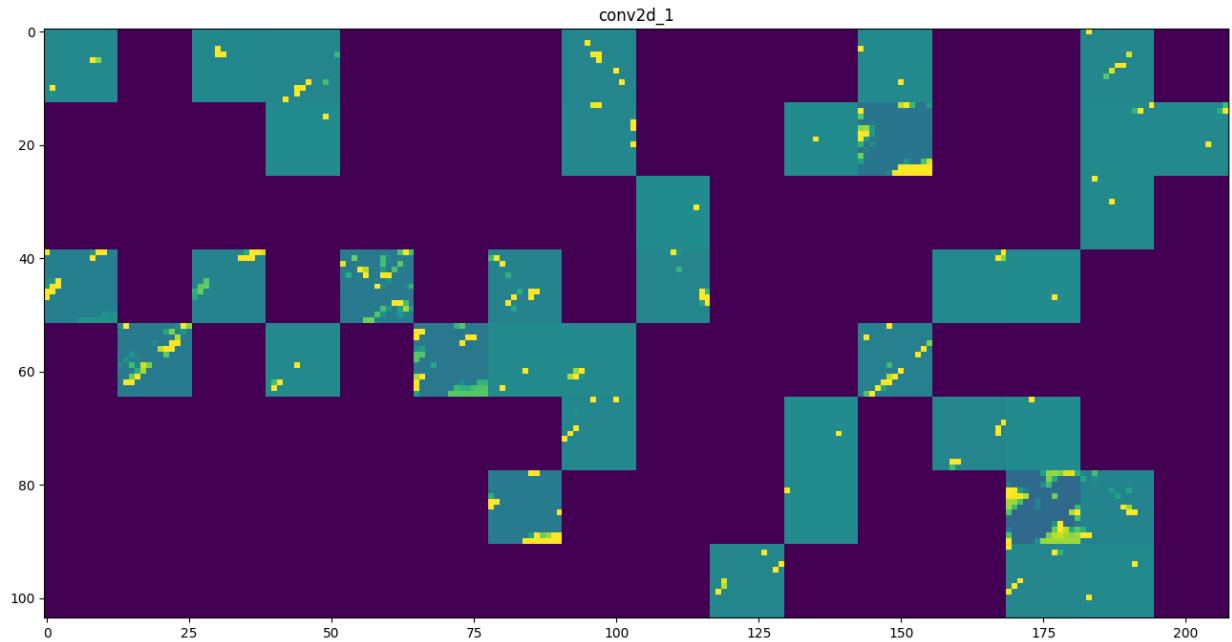
plt.show();
```



1/1 [=====] - 0s 73ms/step

```
<ipython-input-136-9ab60ebbedca>:72: RuntimeWarning: invalid value encountered in divide
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[122]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

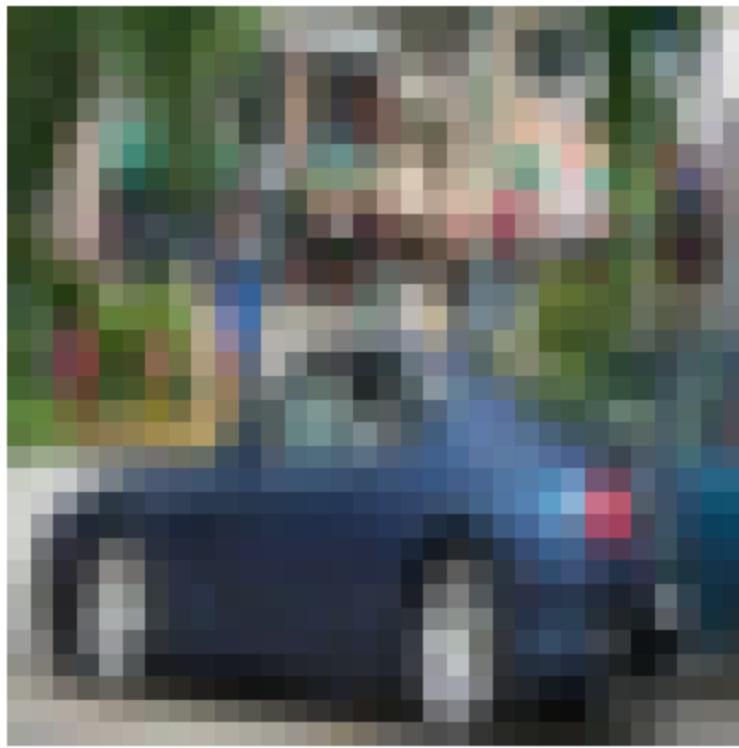
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

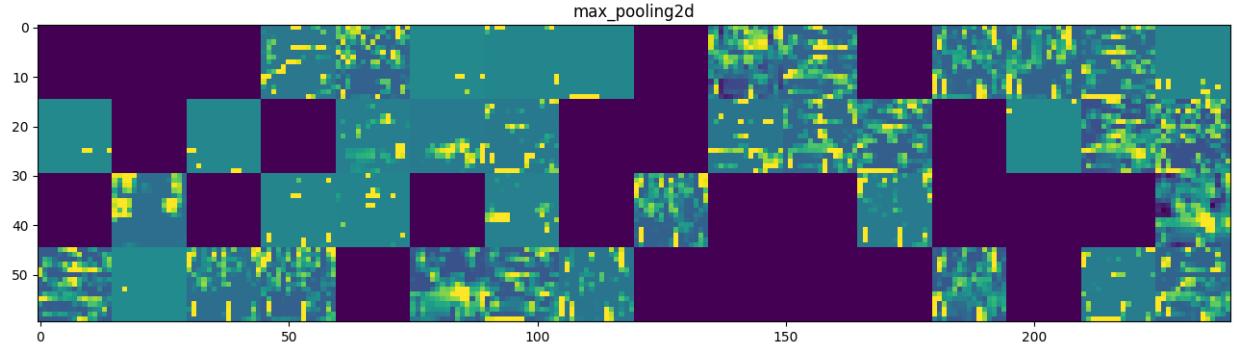
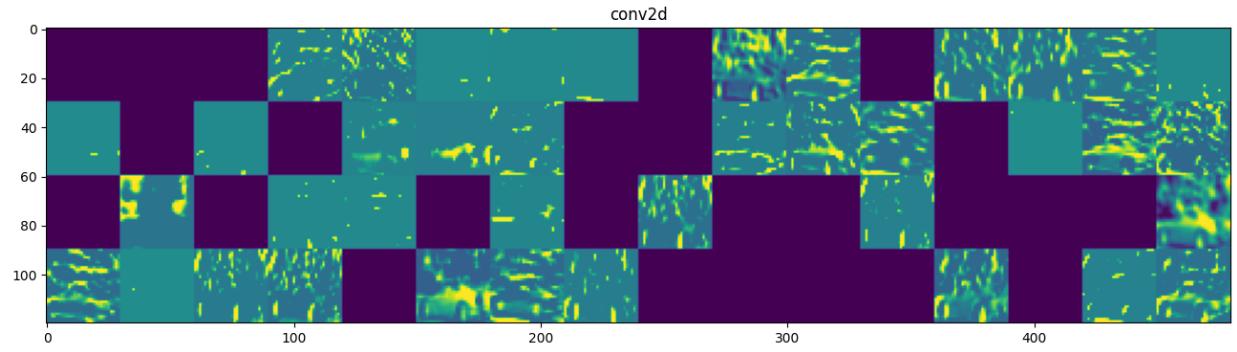
plt.show();
```

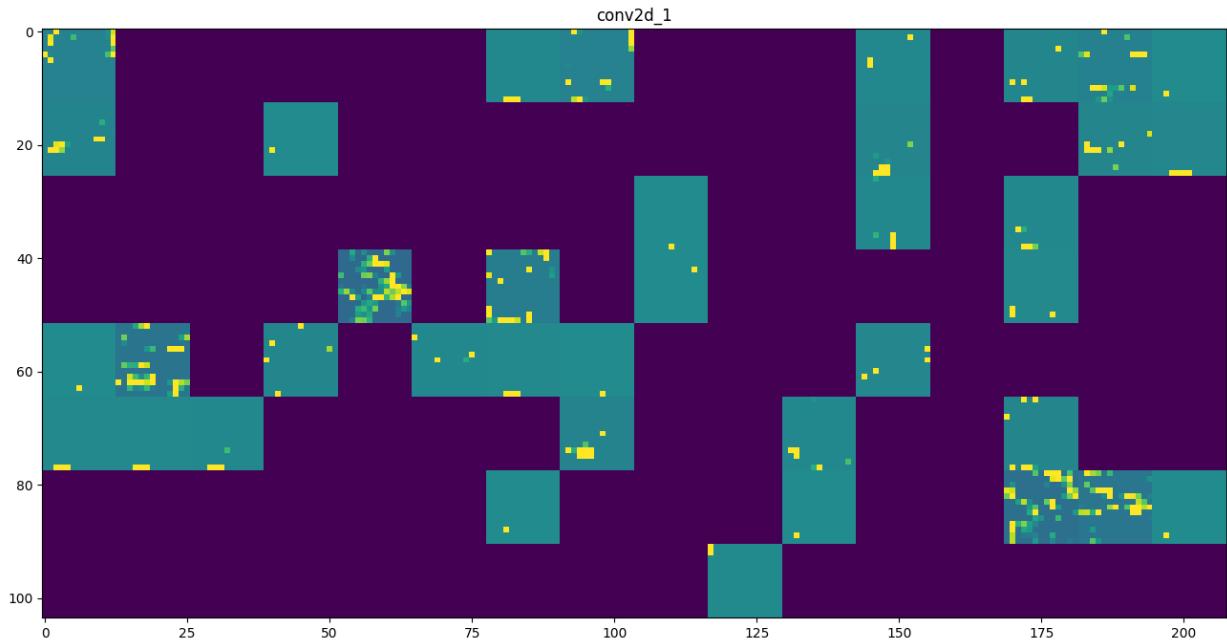


```
1/1 [=====] - 0s 96ms/step
```

```
<ipython-input-137-e0d043c5b9b7>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[75]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

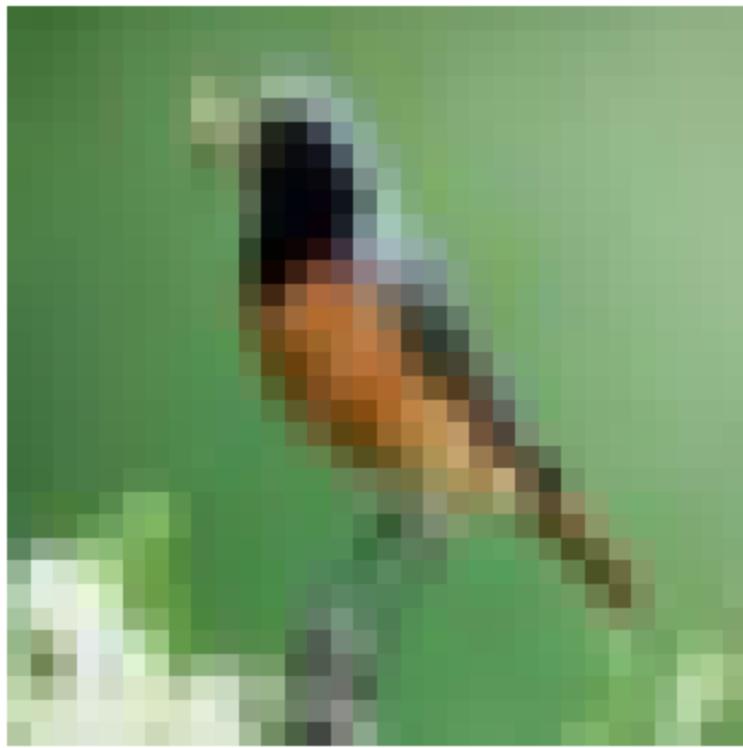
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

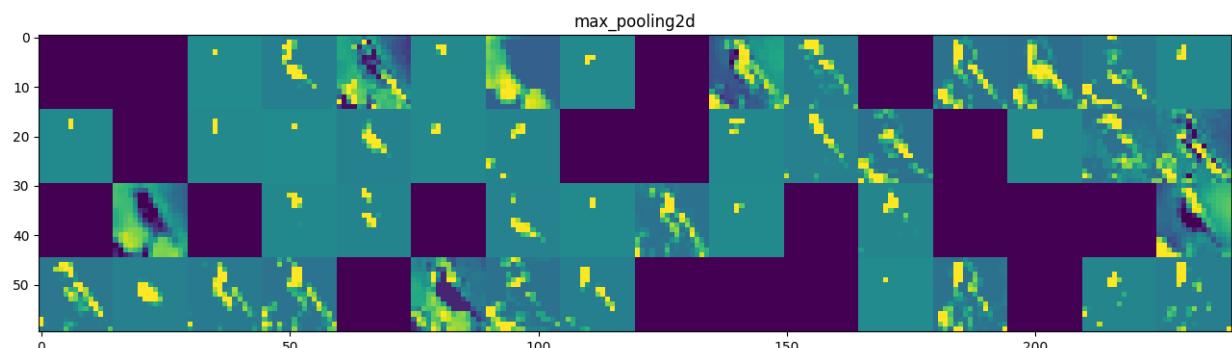
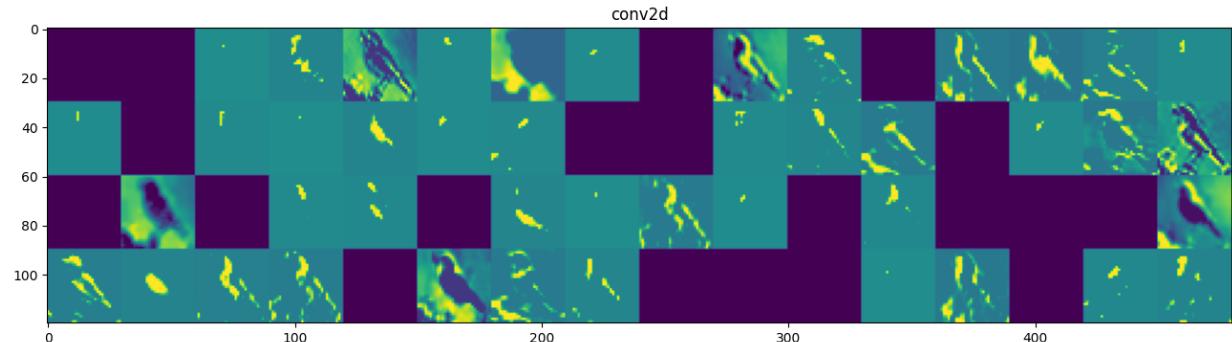
plt.show();
```

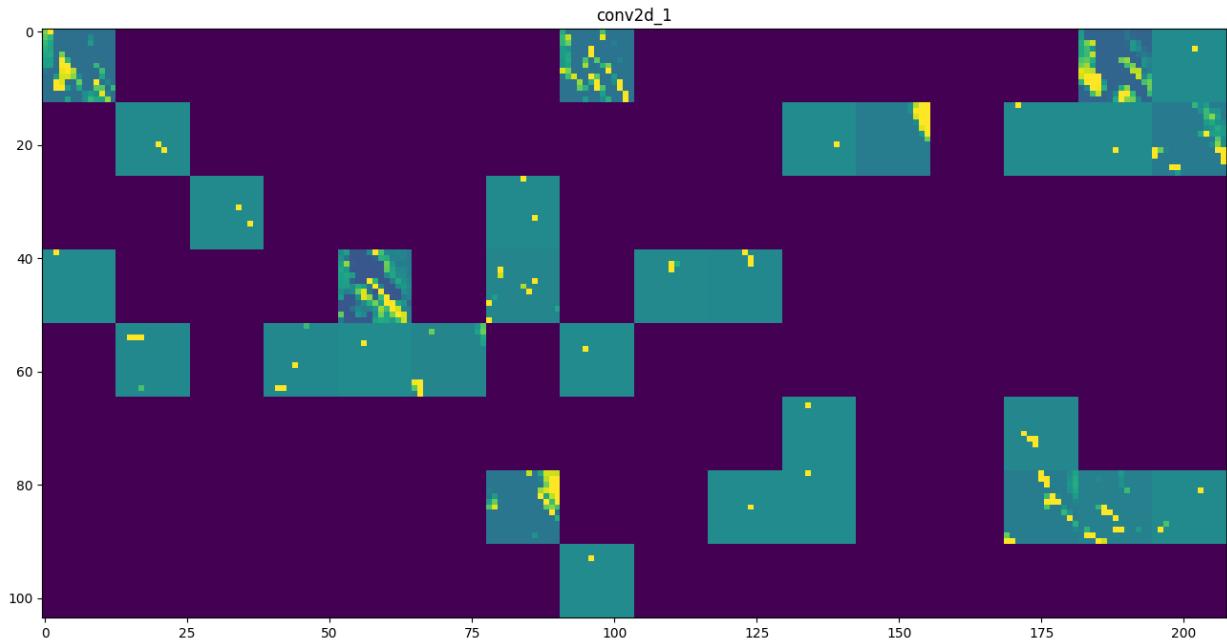


```
WARNING:tensorflow:5 out of the last 317 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7bc7563801f0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for more details.
```

```
1/1 [=====] - 0s 114ms/step
```

```
<ipython-input-138-4848b71b261c>:72: RuntimeWarning: invalid value encountered in divide
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[184]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

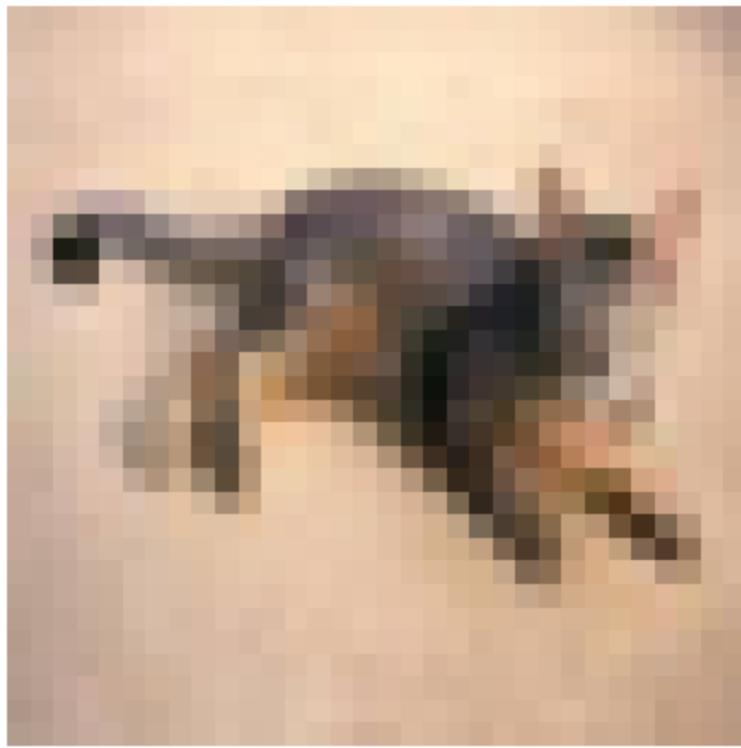
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

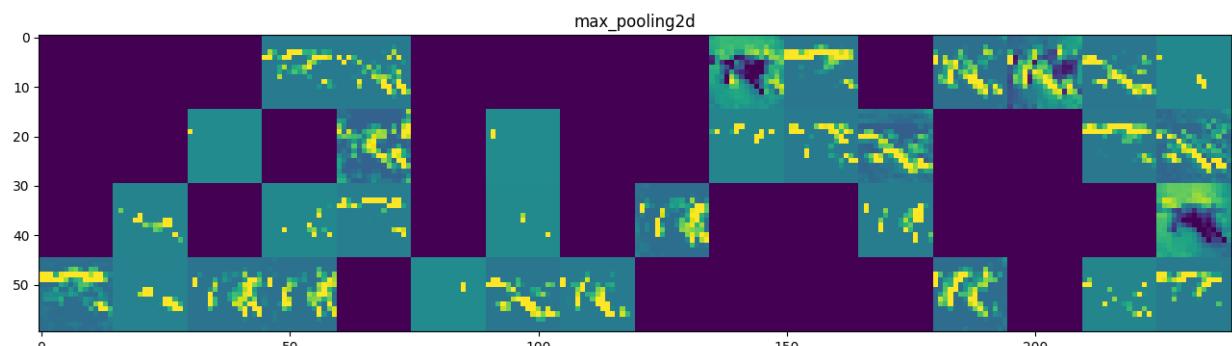
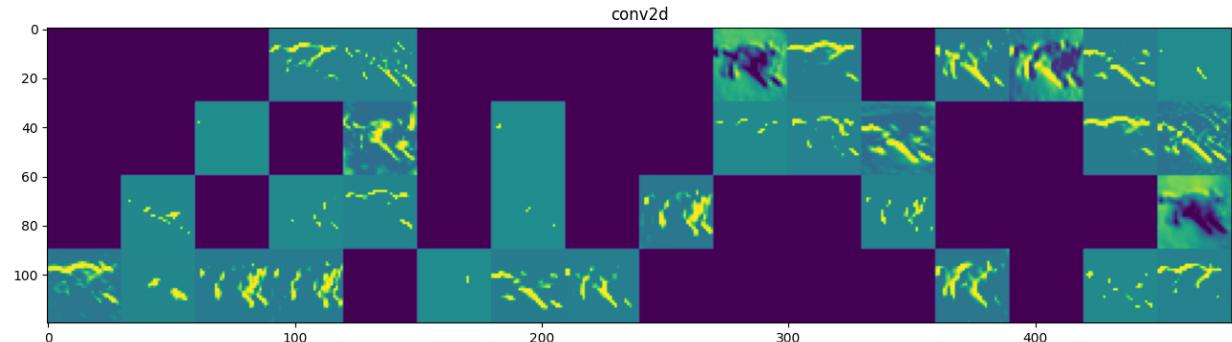
plt.show();
```

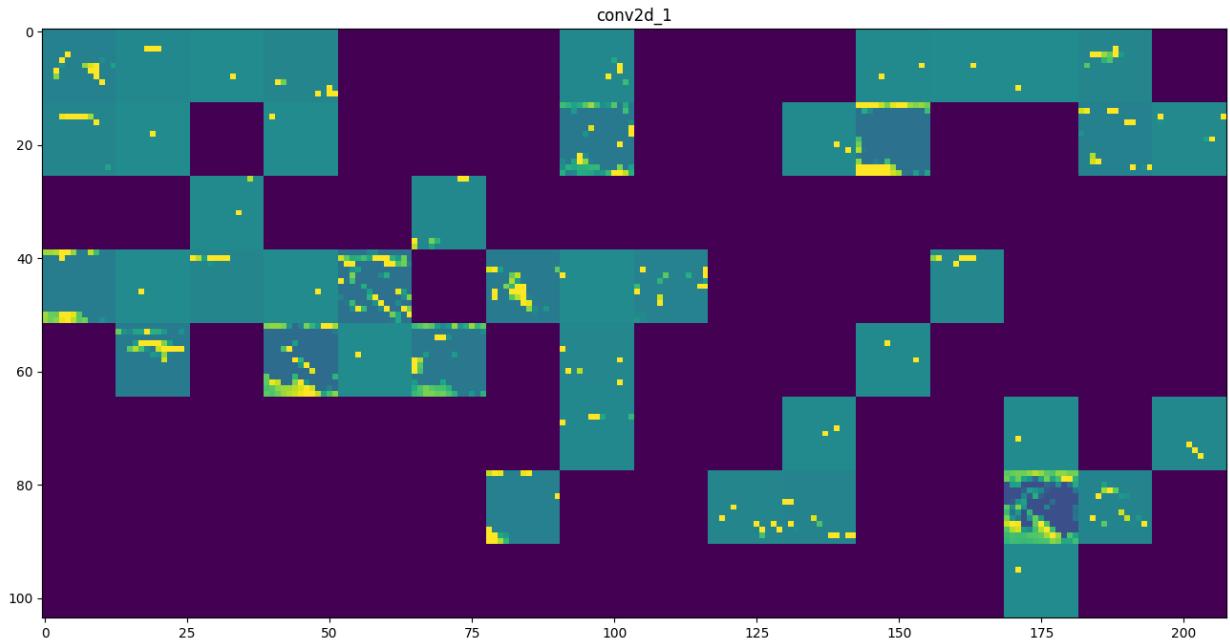


```
WARNING:tensorflow:6 out of the last 318 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7bc756421d80> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for more details.
```

```
1/1 [=====] - 0s 116ms/step
```

```
<ipython-input-139-3bac8bdd9965>:72: RuntimeWarning: invalid value encountered in divide
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[159]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

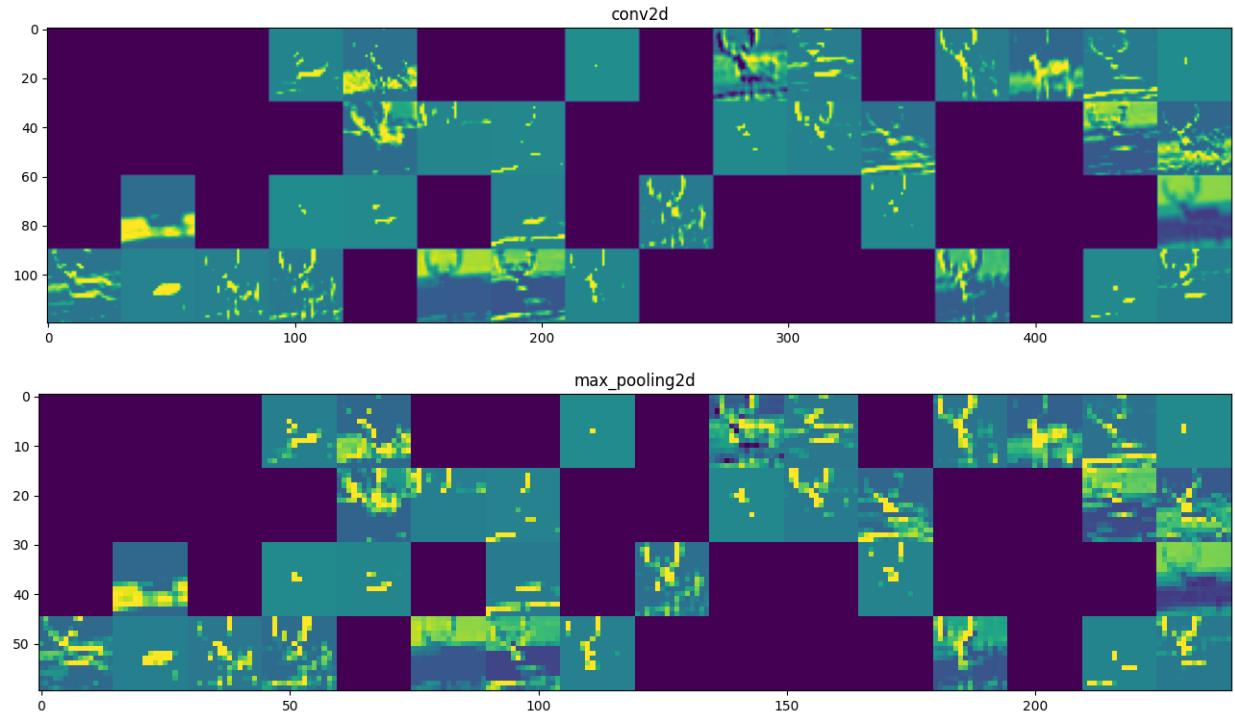
plt.show();
```

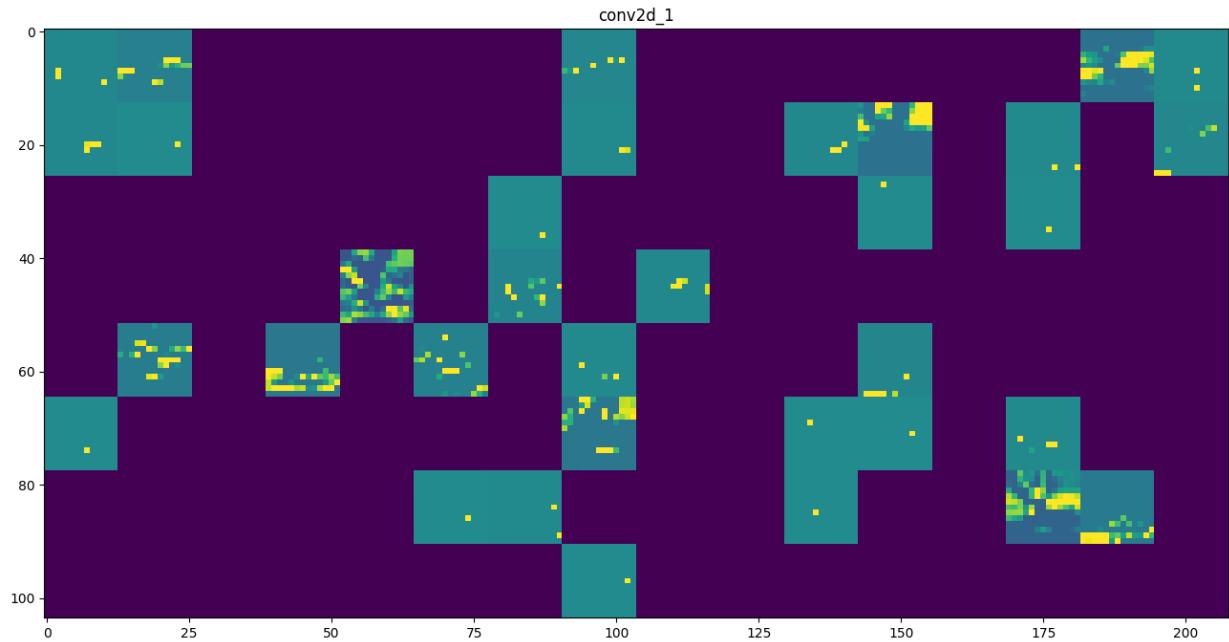


```
1/1 [=====] - 0s 89ms/step
```

```
<ipython-input-140-a52112e96c59>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[24]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

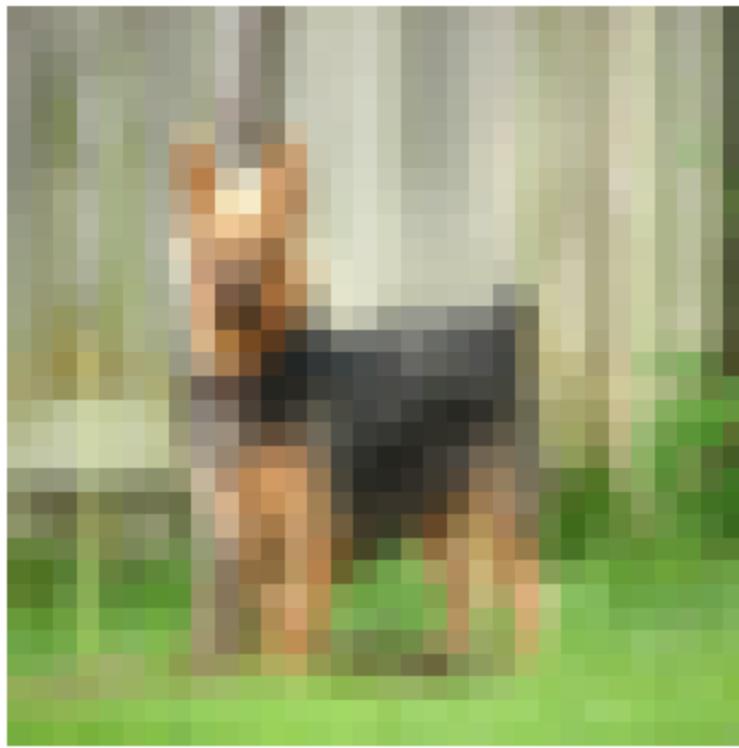
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

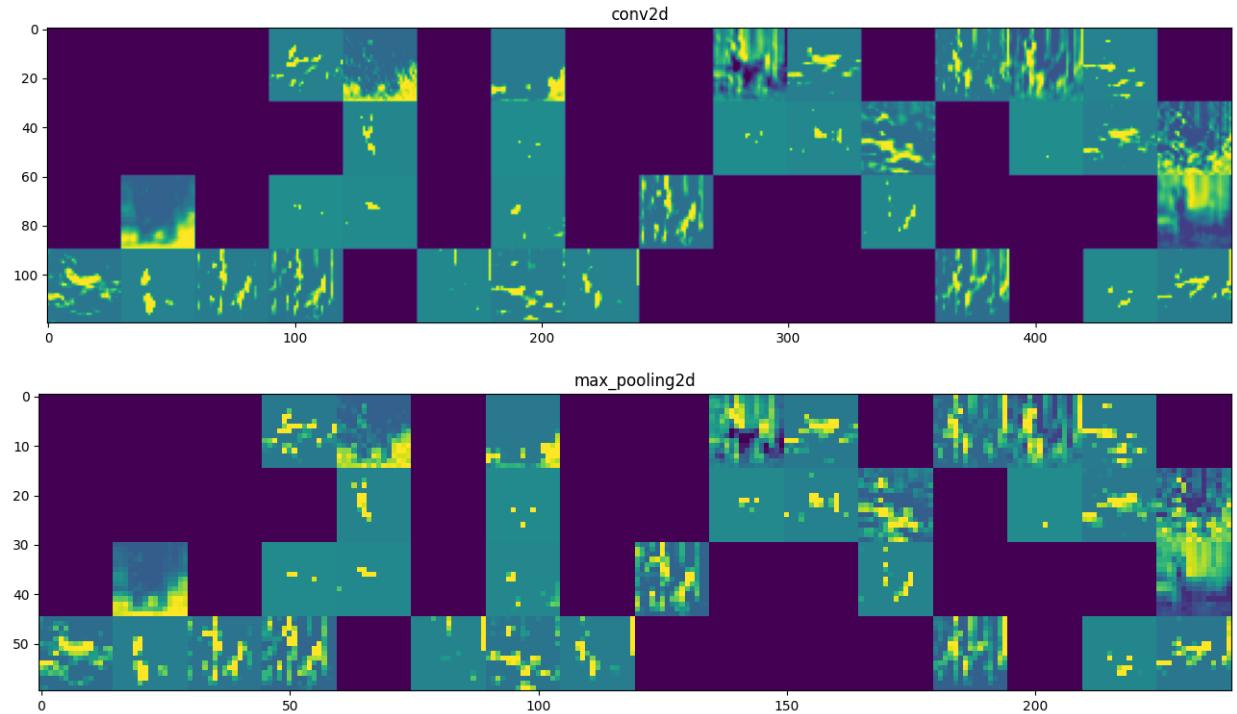
plt.show();
```

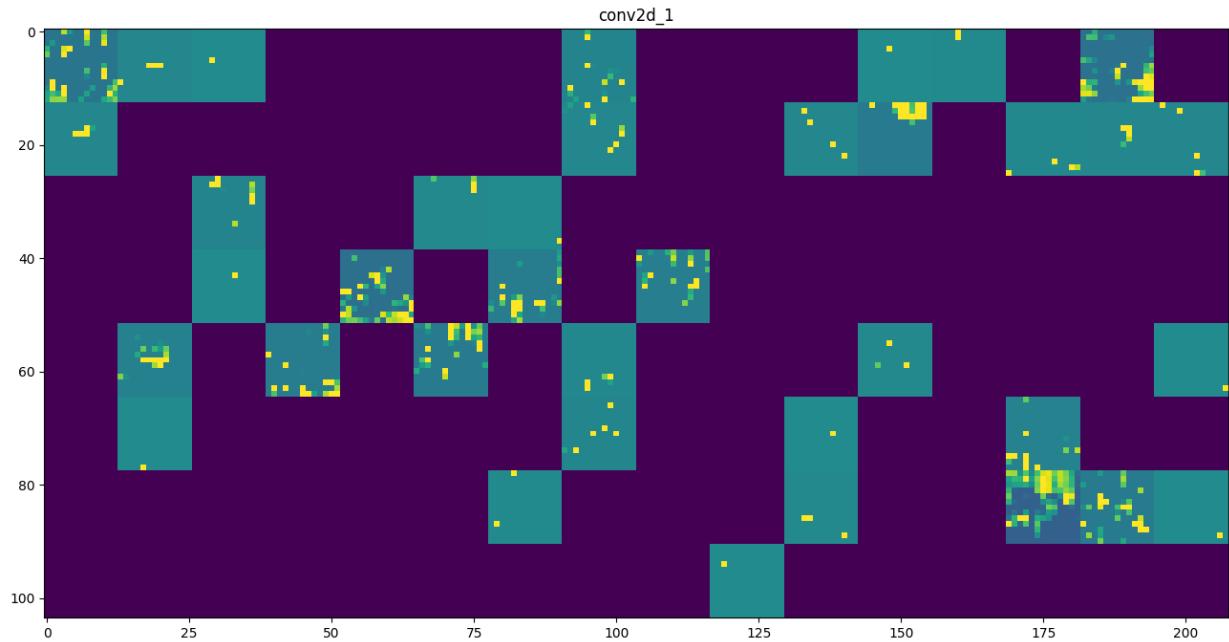


```
1/1 [=====] - 0s 188ms/step
```

```
<ipython-input-141-edee34e9d17b>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[152]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

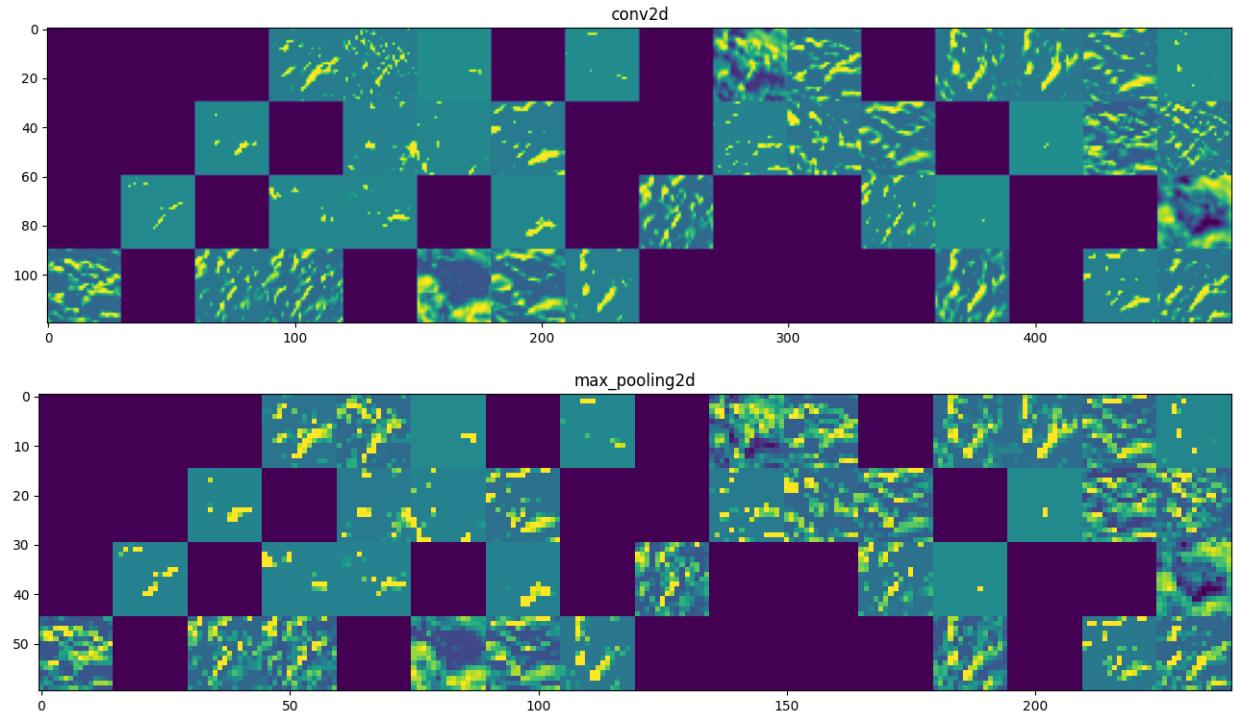
plt.show();
```

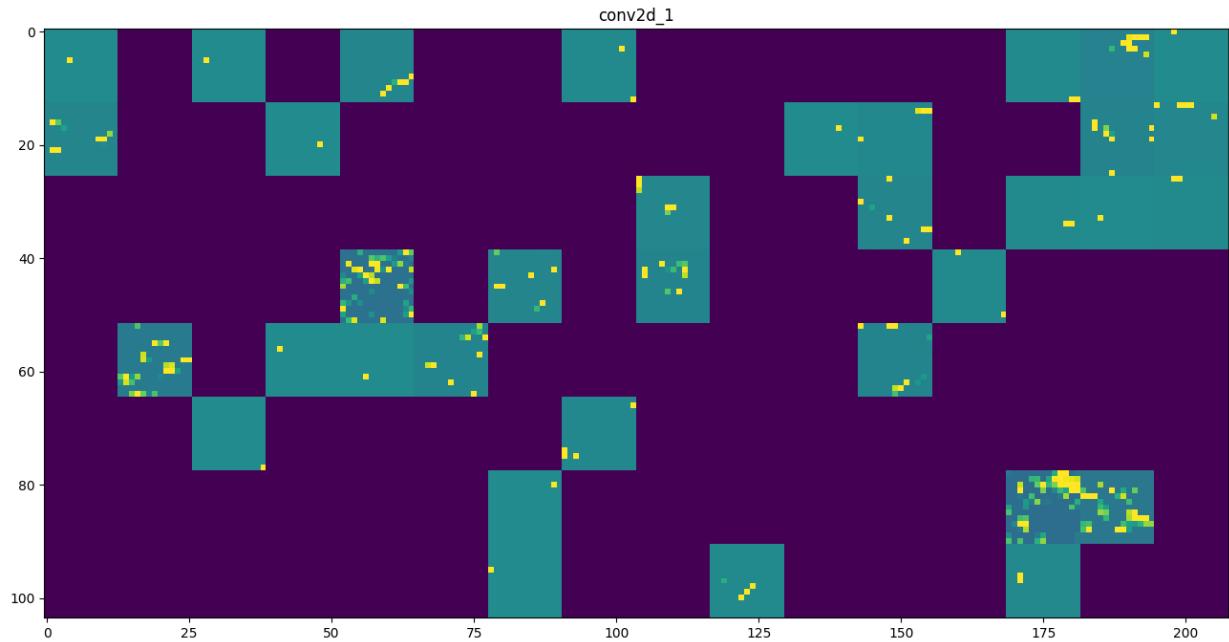


```
1/1 [=====] - 0s 101ms/step
```

```
<ipython-input-142-0ef3f9c61bfb>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[2004]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

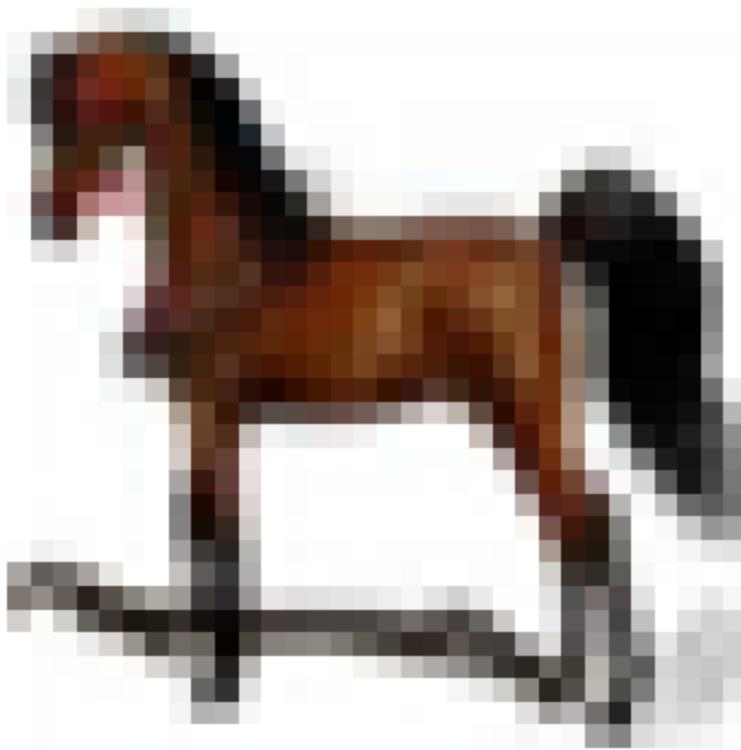
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

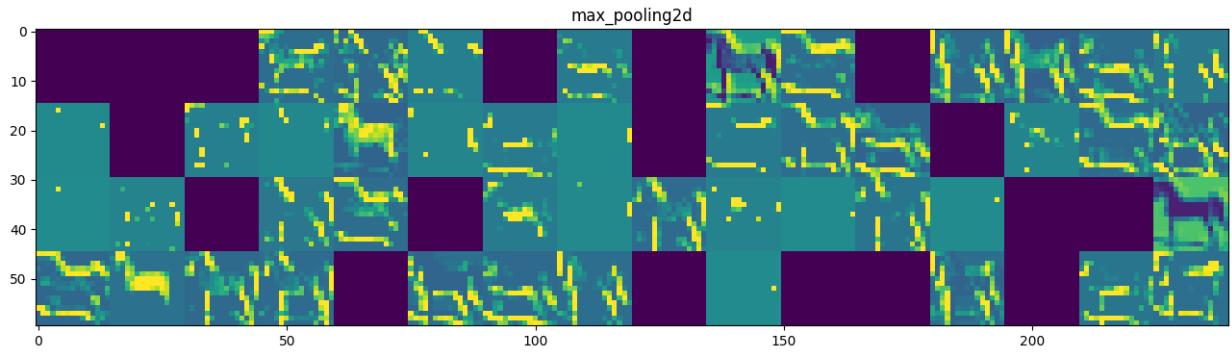
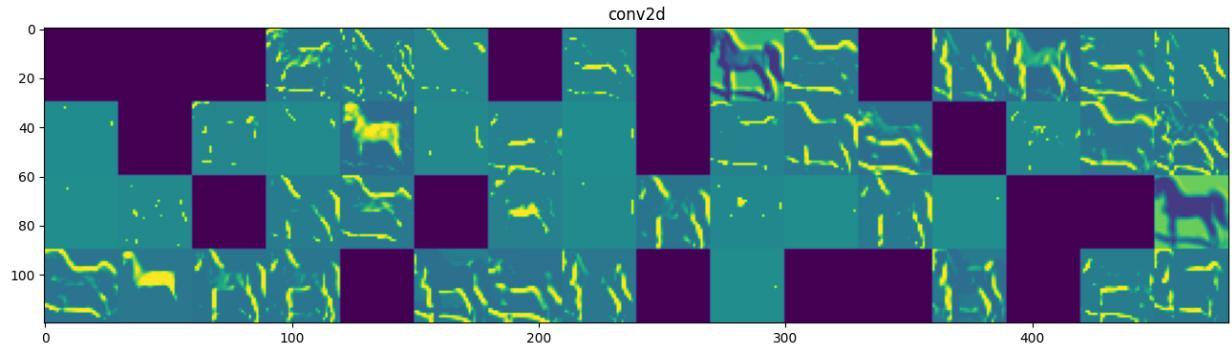
plt.show();
```

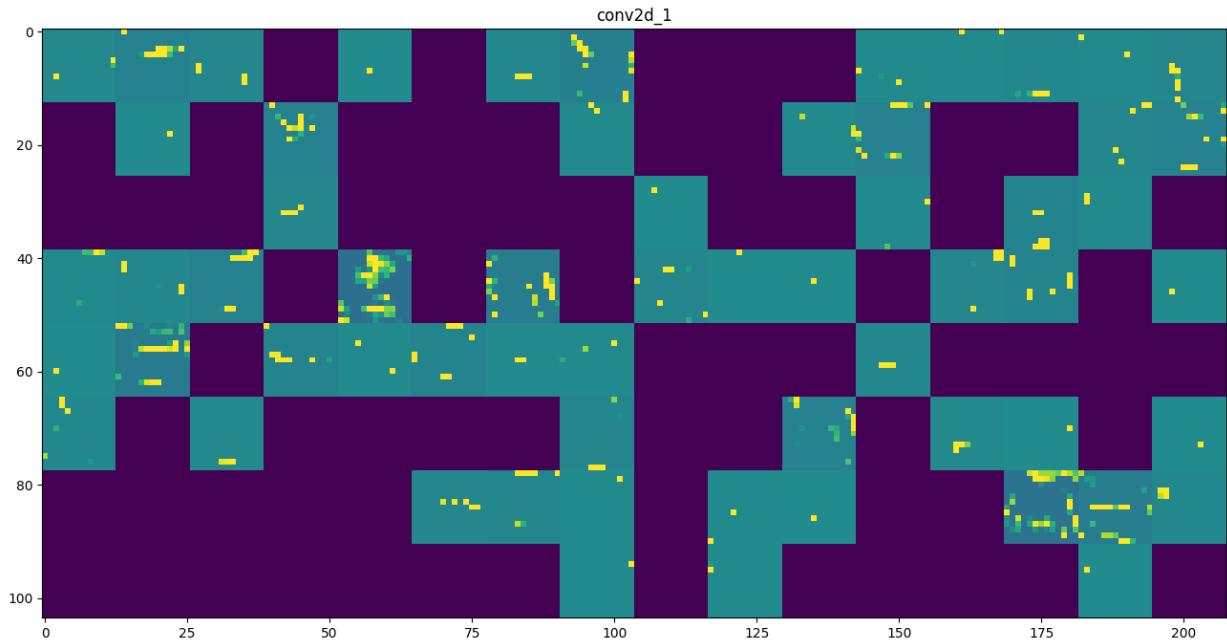


```
1/1 [=====] - 0s 68ms/step
```

```
<ipython-input-143-5c55fce06c66>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[185]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

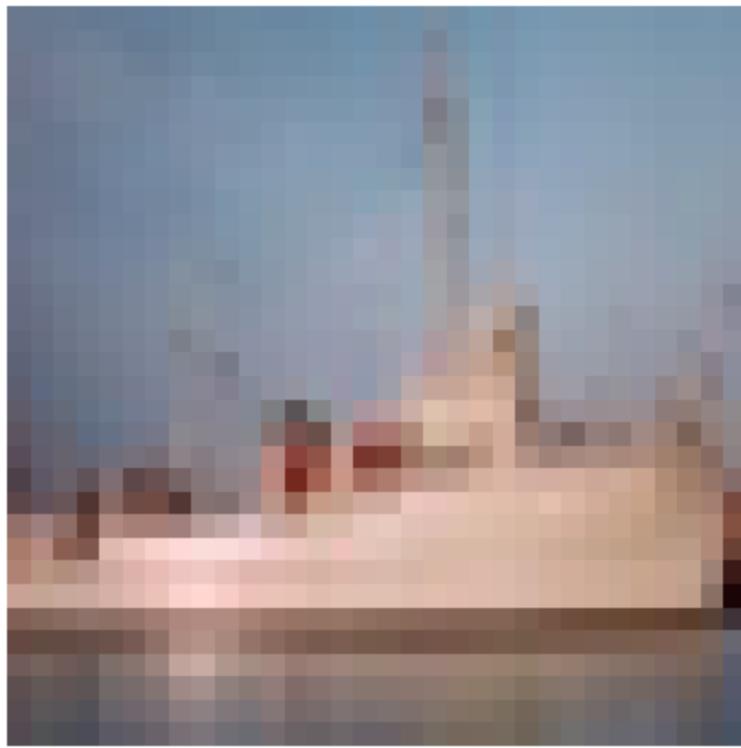
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

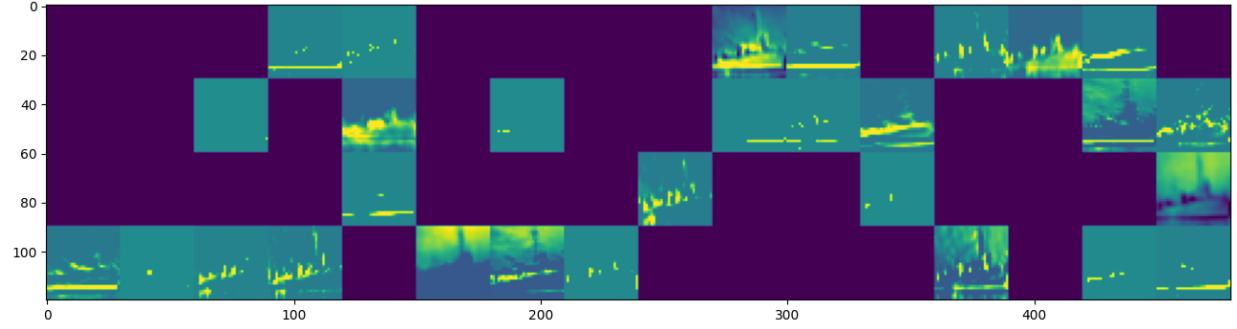


```
1/1 [=====] - 0s 105ms/step
```

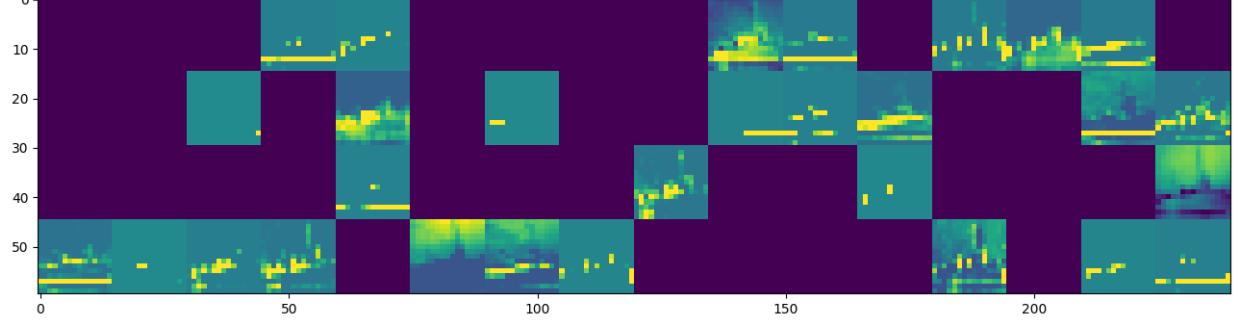
```
<ipython-input-144-f532589aadeb>:72: RuntimeWarning: invalid value encountered in divide
```

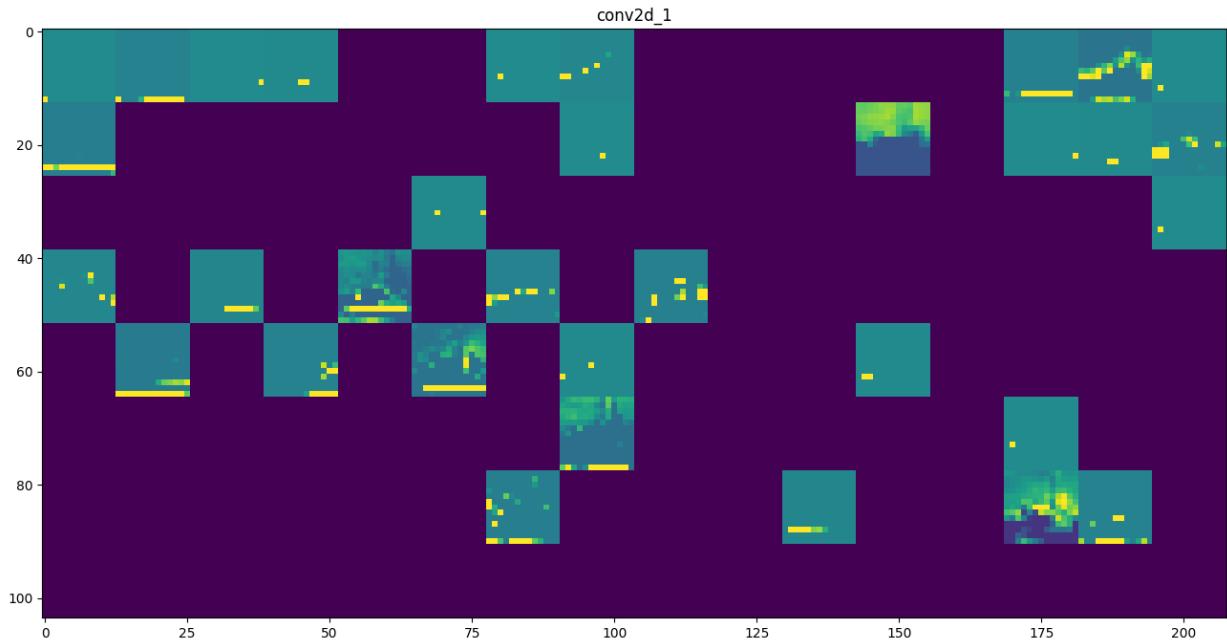
```
    channel_image /= channel_image.std()
```

conv2d



max_pooling2d





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[133]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

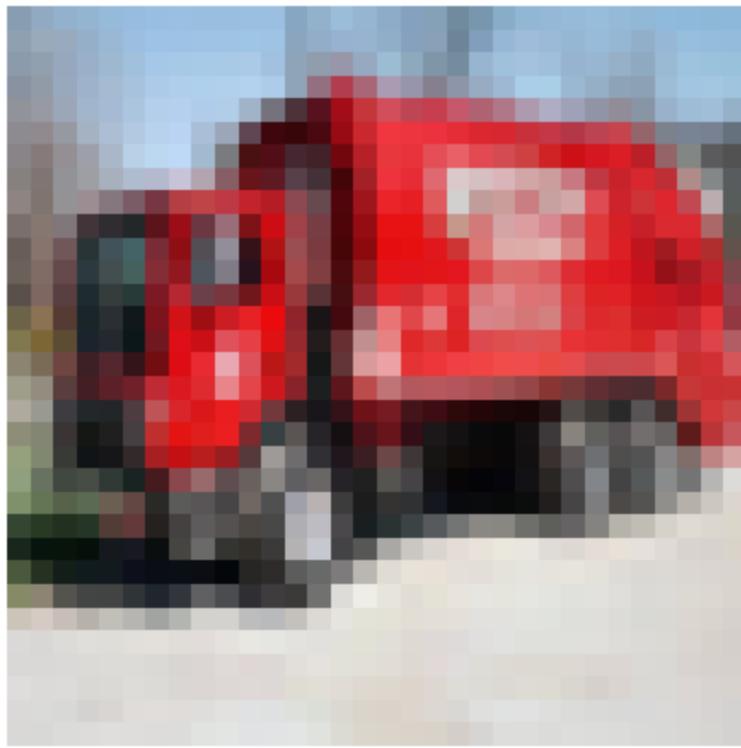
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

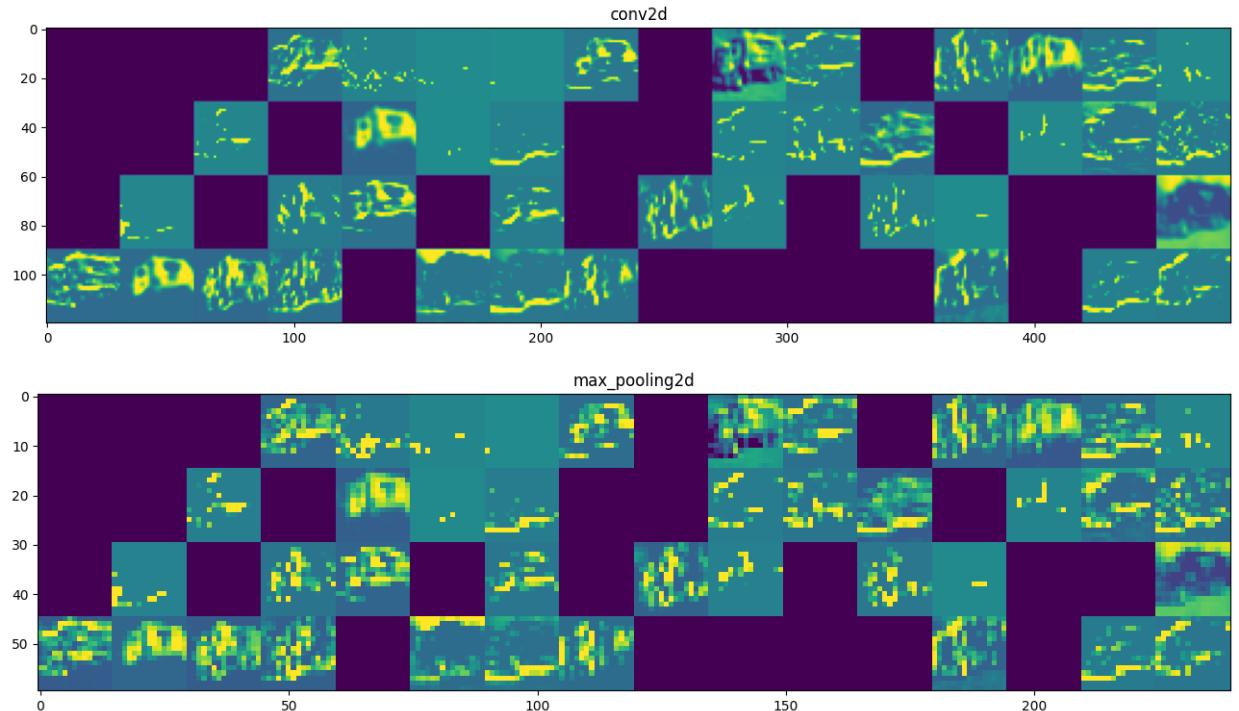
plt.show();
```

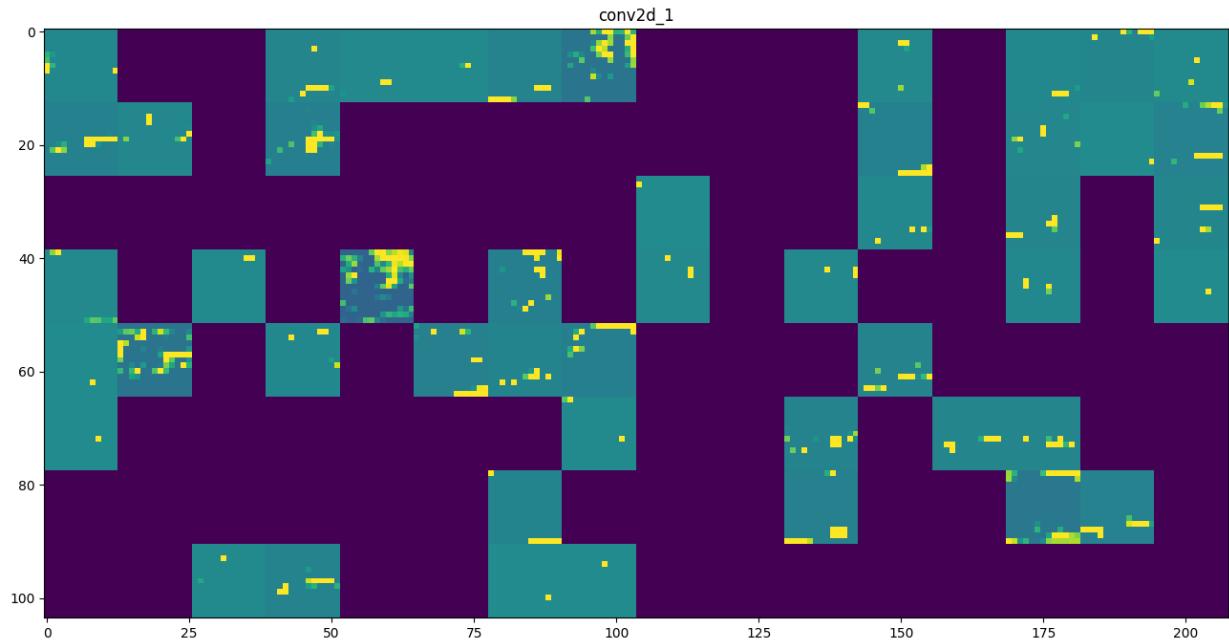


```
1/1 [=====] - 0s 151ms/step
```

```
<ipython-input-145-8e6847644768>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```





5) Model 4 - CNN with 3 Max Pooling / Hidden Layers and No Regularization Methods

5.1) Build The Model

We use a Sequential class defined in Keras to create our model.

```
In [ ]: k.clear_session()
model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu),
    MaxPool2D((2, 2),strides=2),
    Flatten(),
    Dense(units=384,activation=tf.nn.softmax),
    Dense(units=10, activation=tf.nn.softmax)
])
```



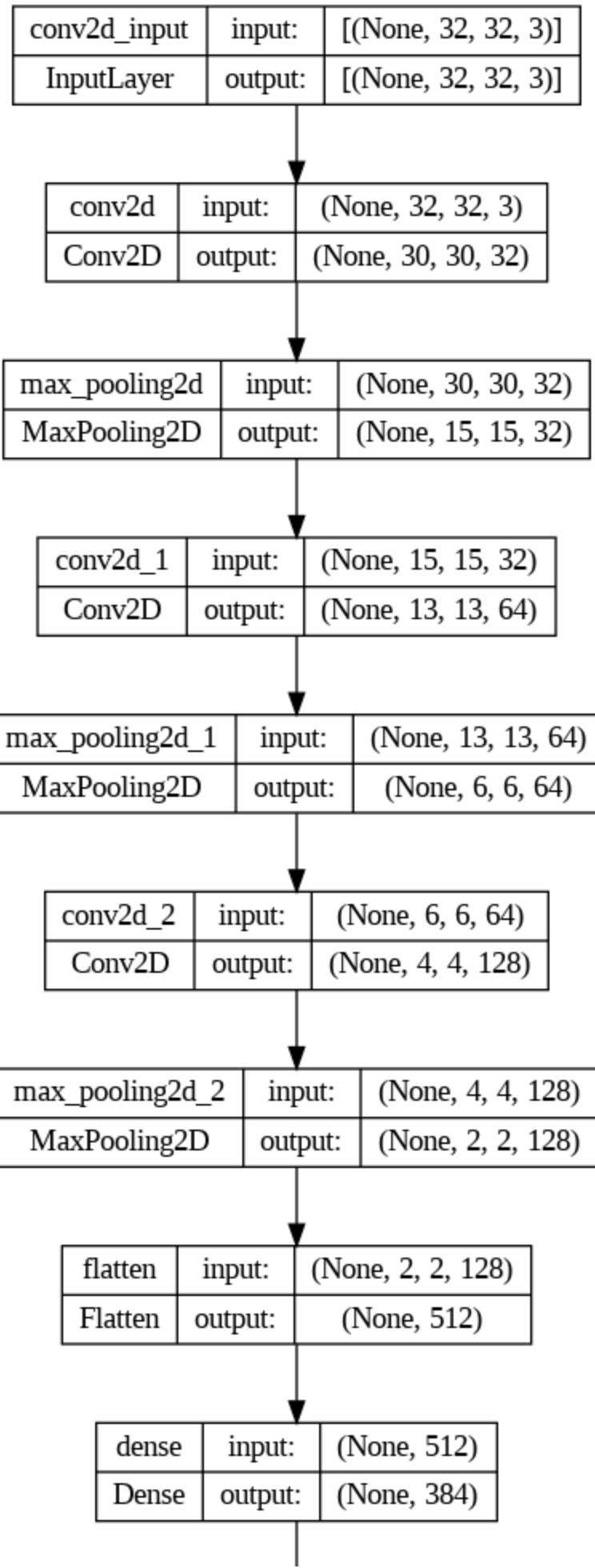
```
In [ ]: model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 384)	196992
dense_1 (Dense)	(None, 10)	3850
<hr/>		
Total params: 294090 (1.12 MB)		
Trainable params: 294090 (1.12 MB)		
Non-trainable params: 0 (0.00 Byte)		

In []: `tf.keras.utils.plot_model(model, "CIFAR10.png", show_shapes=True)`

Out[]:



dense_1	input:	(None, 384)
Dense	output:	(None, 10)

Let's now compile and train the model.

tf.keras.losses.SparseCategoricalCrossentropy

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Module: tf.keras.callbacks

tf.keras.callbacks.EarlyStopping

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

tf.keras.callbacks.ModelCheckpoint

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

```
In [ ]: model.compile(optimizer='adam',
                      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
                      metrics=['accuracy'])
```

```
In [ ]: history = model.fit(x_train_norm
                           ,y_train_split
                           ,epochs=20
                           ,validation_data=(x_valid_norm, y_valid_split)
                           ,callbacks=[
                           tf.keras.callbacks.ModelCheckpoint("Model_4", save_best_only=True,
                           )
                           ])
```

```
Epoch 1/20
1407/1407 [=====] - 40s 28ms/step - loss: 2.1016 - accuracy: 0.2057 - val_loss: 1.9738 - val_accuracy: 0.2310
Epoch 2/20
1407/1407 [=====] - 41s 29ms/step - loss: 1.8828 - accuracy: 0.2676 - val_loss: 1.7826 - val_accuracy: 0.3110
Epoch 3/20
1407/1407 [=====] - 50s 35ms/step - loss: 1.7215 - accuracy: 0.3564 - val_loss: 1.6597 - val_accuracy: 0.3868
Epoch 4/20
1407/1407 [=====] - 41s 29ms/step - loss: 1.5498 - accuracy: 0.4386 - val_loss: 1.4975 - val_accuracy: 0.4672
Epoch 5/20
1407/1407 [=====] - 40s 28ms/step - loss: 1.4101 - accuracy: 0.4953 - val_loss: 1.3904 - val_accuracy: 0.5052
Epoch 6/20
1407/1407 [=====] - 40s 29ms/step - loss: 1.2995 - accuracy: 0.5462 - val_loss: 1.3247 - val_accuracy: 0.5280
Epoch 7/20
1407/1407 [=====] - 39s 28ms/step - loss: 1.2035 - accuracy: 0.5806 - val_loss: 1.2753 - val_accuracy: 0.5500
Epoch 8/20
1407/1407 [=====] - 39s 28ms/step - loss: 1.1327 - accuracy: 0.6038 - val_loss: 1.2303 - val_accuracy: 0.5634
Epoch 9/20
1407/1407 [=====] - 40s 28ms/step - loss: 1.0677 - accuracy: 0.6282 - val_loss: 1.1809 - val_accuracy: 0.5856
Epoch 10/20
1407/1407 [=====] - 43s 31ms/step - loss: 1.0130 - accuracy: 0.6468 - val_loss: 1.1761 - val_accuracy: 0.5926
Epoch 11/20
1407/1407 [=====] - 39s 28ms/step - loss: 0.9642 - accuracy: 0.6638 - val_loss: 1.2075 - val_accuracy: 0.5664
Epoch 12/20
1407/1407 [=====] - 40s 28ms/step - loss: 0.9192 - accuracy: 0.6793 - val_loss: 1.1656 - val_accuracy: 0.5906
Epoch 13/20
1407/1407 [=====] - 39s 28ms/step - loss: 0.8771 - accuracy: 0.6941 - val_loss: 1.1442 - val_accuracy: 0.5948
Epoch 14/20
1407/1407 [=====] - 39s 28ms/step - loss: 0.8376 - accuracy: 0.7081 - val_loss: 1.1475 - val_accuracy: 0.5980
Epoch 15/20
1407/1407 [=====] - 40s 28ms/step - loss: 0.8044 - accuracy: 0.7192 - val_loss: 1.1581 - val_accuracy: 0.6046
Epoch 16/20
1407/1407 [=====] - 39s 28ms/step - loss: 0.7725 - accuracy: 0.7308 - val_loss: 1.1552 - val_accuracy: 0.6094
Epoch 17/20
1407/1407 [=====] - 39s 28ms/step - loss: 0.7366 - accuracy: 0.7422 - val_loss: 1.1733 - val_accuracy: 0.6100
Epoch 18/20
1407/1407 [=====] - 39s 28ms/step - loss: 0.7162 - accuracy: 0.7499 - val_loss: 1.1977 - val_accuracy: 0.6060
Epoch 19/20
1407/1407 [=====] - 39s 27ms/step - loss: 0.6848 - accuracy: 0.7608 - val_loss: 1.1958 - val_accuracy: 0.6078
Epoch 20/20
1407/1407 [=====] - 39s 28ms/step - loss: 0.6633 - accuracy: 0.7660 - val_loss: 1.2283 - val_accuracy: 0.6066
```

5.2) Evaluate Model Performance

```
In [ ]: model = tf.keras.models.load_model("Model_4")
print(f"Training accuracy: {model.evaluate(x_train_norm, y_train_split)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_norm, y_valid_split)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_norm, y_test)[1]:.3f}")

1407/1407 [=====] - 12s 9ms/step - loss: 0.8404 - accuracy: 0.7131
Training accuracy: 0.713
157/157 [=====] - 1s 8ms/step - loss: 1.1442 - accuracy: 0.5948
Validation accuracy: 0.595
313/313 [=====] - 3s 11ms/step - loss: 1.1962 - accuracy: 0.5850
Test accuracy: 0.585
```

```
In [ ]: preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)

313/313 [=====] - 3s 9ms/step
shape of preds: (10000, 10)
```

```
In [ ]: history_dict = history.history
history_dict.keys()

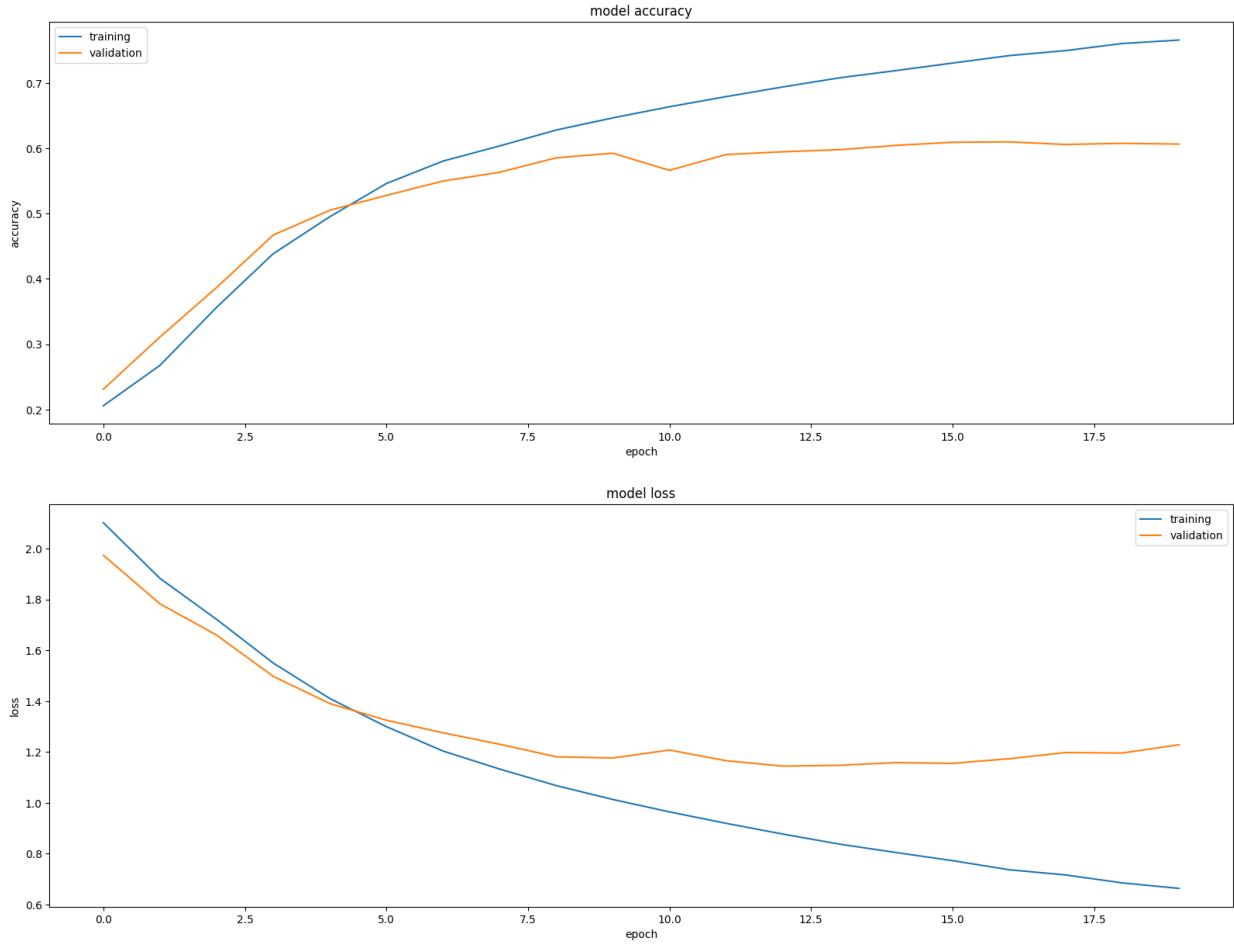
Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)
```

	loss	accuracy	val_loss	val_accuracy
15	0.773	0.731	1.155	0.609
16	0.737	0.742	1.173	0.610
17	0.716	0.750	1.198	0.606
18	0.685	0.761	1.196	0.608
19	0.663	0.766	1.228	0.607

```
In [ ]: plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',

<ipython-input-8-353fbbae40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.
    ax = plt.subplot(subplot)
```



Let's examine the precision, recall, F1 score, and confusion matrix.

```
In [ ]: pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

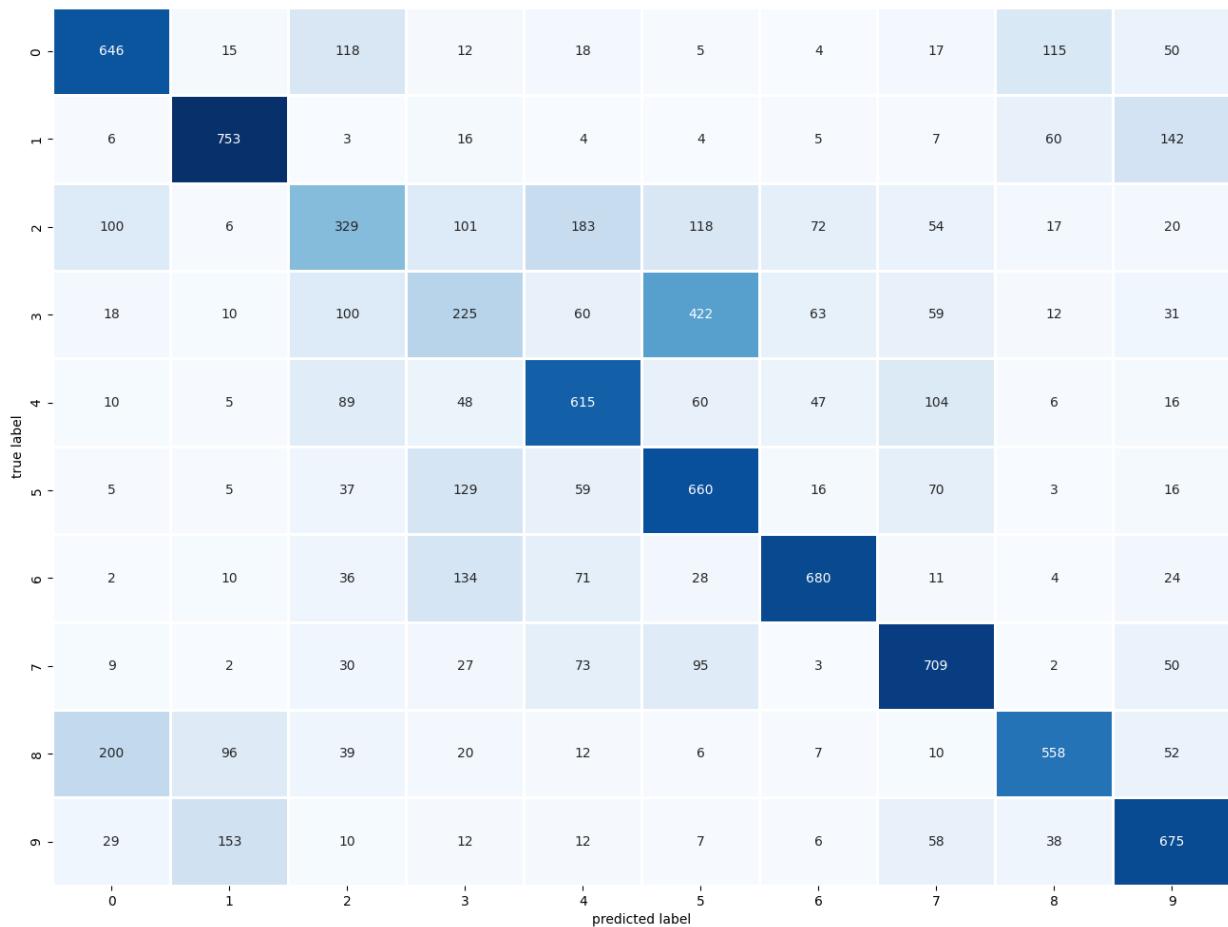
313/313 [=====] - 3s 9ms/step
```

```
In [ ]: print_validation_report(y_test, pred1)
```

Classification Report				
	precision	recall	f1-score	support
0	0.63	0.65	0.64	1000
1	0.71	0.75	0.73	1000
2	0.42	0.33	0.37	1000
3	0.31	0.23	0.26	1000
4	0.56	0.61	0.58	1000
5	0.47	0.66	0.55	1000
6	0.75	0.68	0.71	1000
7	0.65	0.71	0.68	1000
8	0.68	0.56	0.61	1000
9	0.63	0.68	0.65	1000
accuracy			0.58	10000
macro avg	0.58	0.58	0.58	10000
weighted avg	0.58	0.58	0.58	10000

Accuracy Score: 0.585

Root Mean Square Error: 2.836635330810078

In []: `plot_confusion_matrix(y_test,pred1)`

Load HDF5 Model Format

`tf.keras.models.load_model`https://www.tensorflow.org/api_docs/python/tf/keras/models/load_modelIn []: `model = tf.keras.models.load_model('Model_4')
preds = model.predict(x_test_norm)
preds.shape`

313/313 [=====] - 3s 9ms/step

Out[]:

Let's examine the predictions for the testing dataset.

In []: `cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)

df = pd.DataFrame(preds[0:20], columns = ['airplane'
,'automobile'
, 'bird'
, 'cat'
, 'deer'
, 'dog'
, 'frog'
, 'horse'`

```

        , 'ship'
        , 'truck'])
df.style.format("{:.2%}").background_gradient(cmap=cm)

```

Out[]:

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0	0.05%	0.03%	4.36%	36.08%	0.91%	56.46%	0.64%	1.33%	0.11%	0.04%
1	45.46%	0.14%	4.63%	0.58%	0.36%	0.09%	0.09%	0.08%	48.06%	0.51%
2	0.23%	66.61%	0.01%	0.19%	0.02%	0.04%	0.04%	0.00%	15.72%	17.14%
3	33.66%	0.58%	3.66%	0.95%	0.44%	0.16%	0.17%	0.10%	58.77%	1.51%
4	0.02%	0.06%	7.22%	0.56%	85.94%	0.29%	5.56%	0.33%	0.02%	0.01%
5	0.10%	1.60%	1.30%	5.38%	0.26%	0.49%	89.68%	0.10%	0.09%	0.99%
6	0.91%	46.54%	0.34%	3.69%	0.36%	1.22%	0.59%	0.15%	14.80%	31.40%
7	0.25%	0.82%	9.10%	11.76%	3.84%	2.25%	70.80%	0.57%	0.20%	0.42%
8	0.25%	0.04%	12.47%	35.92%	2.45%	44.68%	0.98%	2.84%	0.29%	0.08%
9	0.18%	65.73%	0.01%	0.24%	0.03%	0.05%	0.06%	0.01%	8.43%	25.25%
10	0.39%	0.08%	23.78%	2.13%	67.27%	1.16%	3.49%	1.40%	0.25%	0.04%
11	0.31%	44.58%	0.03%	0.33%	0.06%	0.11%	0.05%	0.03%	6.68%	47.82%
12	0.71%	1.32%	12.34%	40.21%	6.29%	24.09%	8.42%	3.69%	1.15%	1.78%
13	0.16%	0.00%	0.18%	0.18%	1.06%	0.44%	0.01%	86.05%	0.00%	11.93%
14	0.31%	44.38%	0.03%	0.34%	0.07%	0.11%	0.06%	0.03%	6.52%	48.17%
15	1.61%	0.39%	32.76%	3.64%	48.55%	1.13%	9.26%	1.02%	1.45%	0.18%
16	0.03%	0.02%	3.50%	33.63%	0.69%	60.34%	0.49%	1.23%	0.06%	0.02%
17	1.74%	1.44%	3.81%	6.37%	12.55%	6.53%	0.76%	31.48%	0.66%	34.67%
18	1.16%	39.51%	0.09%	0.61%	0.13%	0.16%	0.11%	0.04%	20.34%	37.84%
19	0.12%	0.98%	2.45%	9.69%	0.48%	1.25%	83.91%	0.24%	0.08%	0.79%

In []:

```

(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[98]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
               , 'automobile'
               , 'bird'
               , 'cat'
               , 'deer'
               , 'dog'
               , 'frog'
               , 'horse'
               , 'ship'
               , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')

```

```
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

# Display the grid
```

```
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                    scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

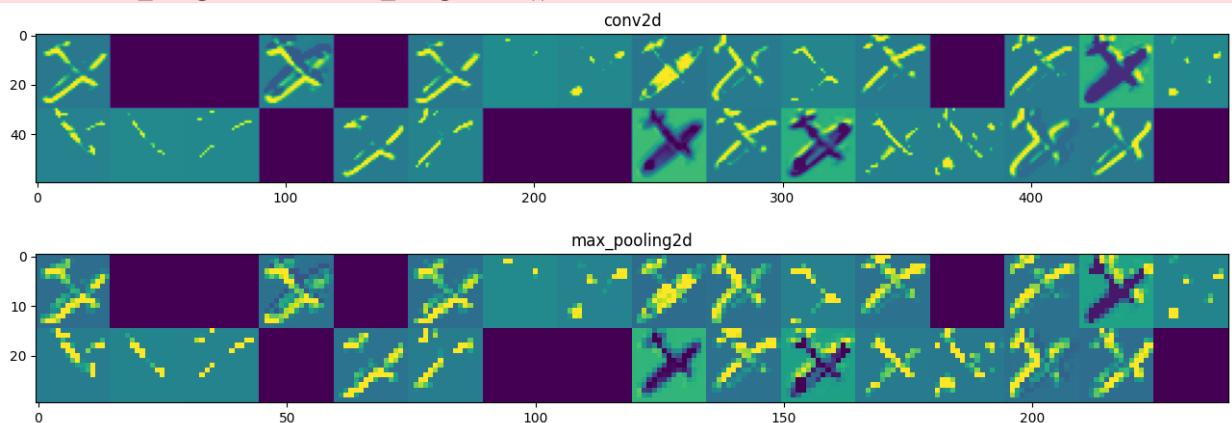
plt.show();
```

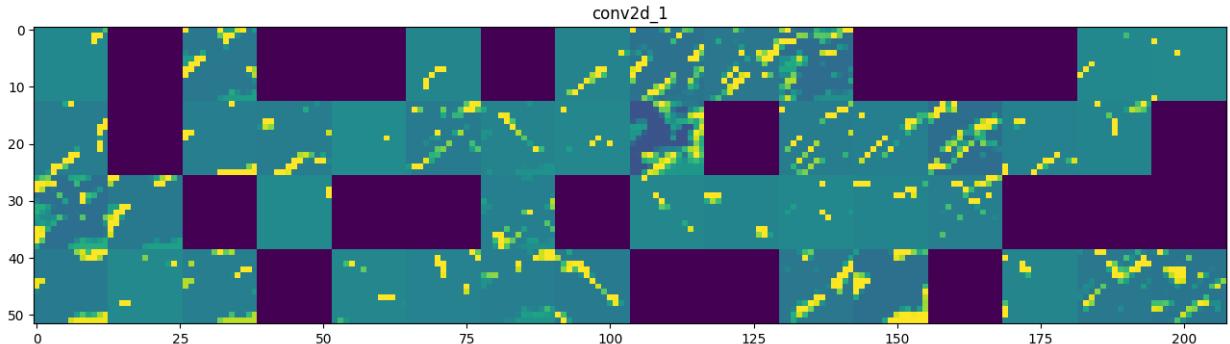


1/1 [=====] - 0s 86ms/step

<ipython-input-161-9ab60ebbedca>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[122]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)
```

```
images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

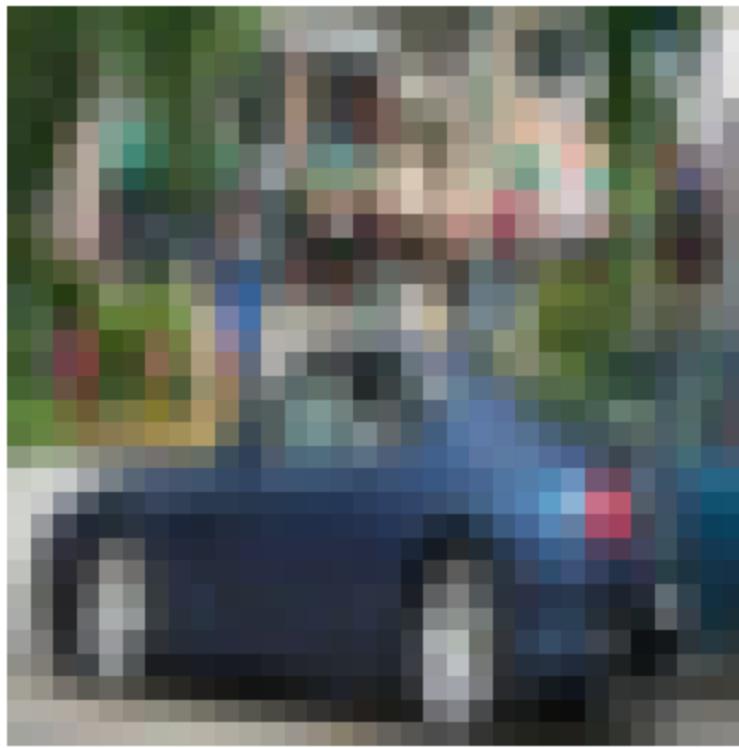
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

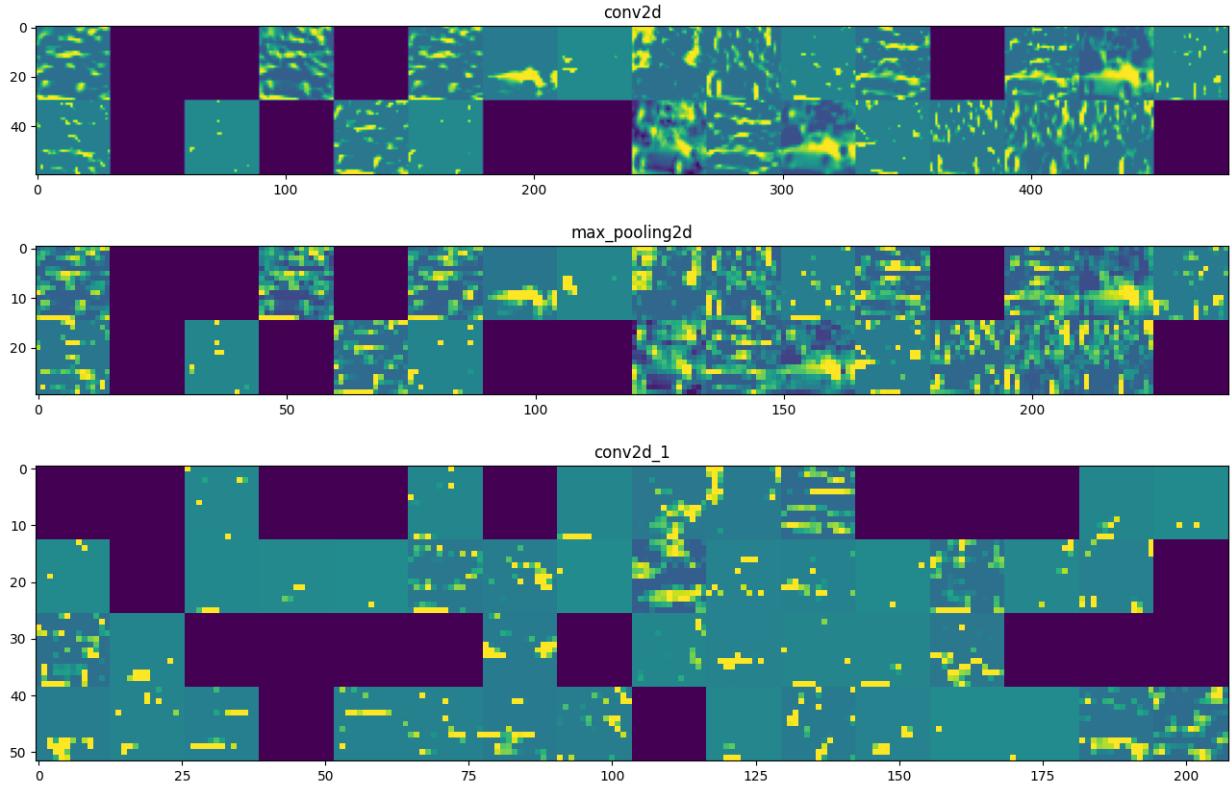
plt.show();
```



```
1/1 [=====] - 0s 80ms/step
```

```
<ipython-input-162-e0d043c5b9b7>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[75]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

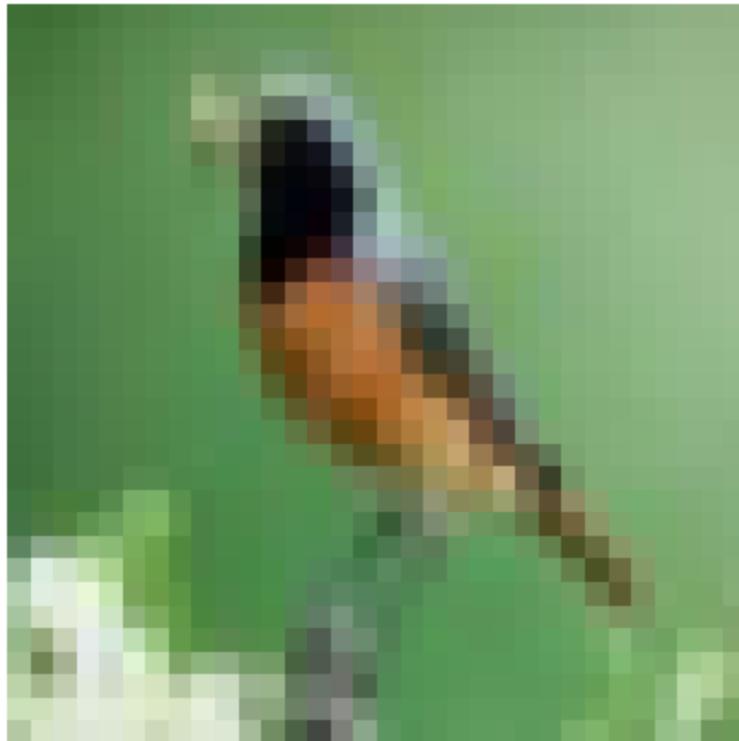
channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

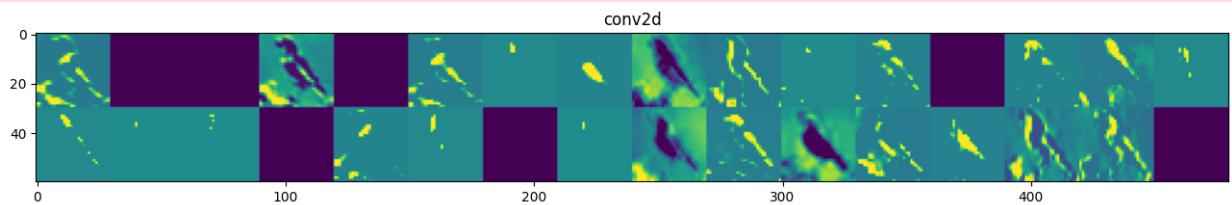
```

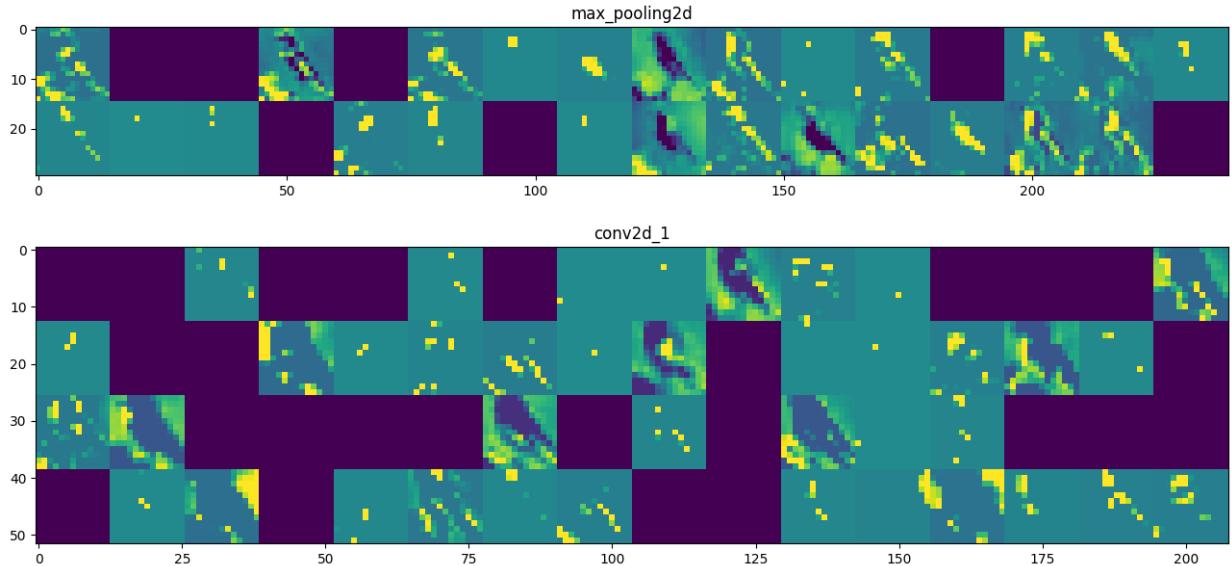


1/1 [=====] - 0s 78ms/step

<ipython-input-163-4848b71b261c>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[184]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

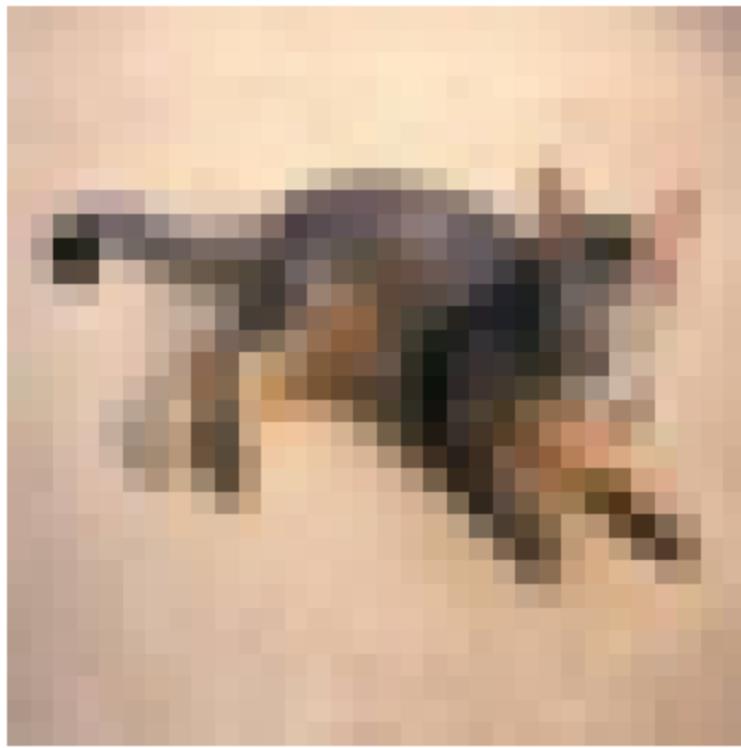
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

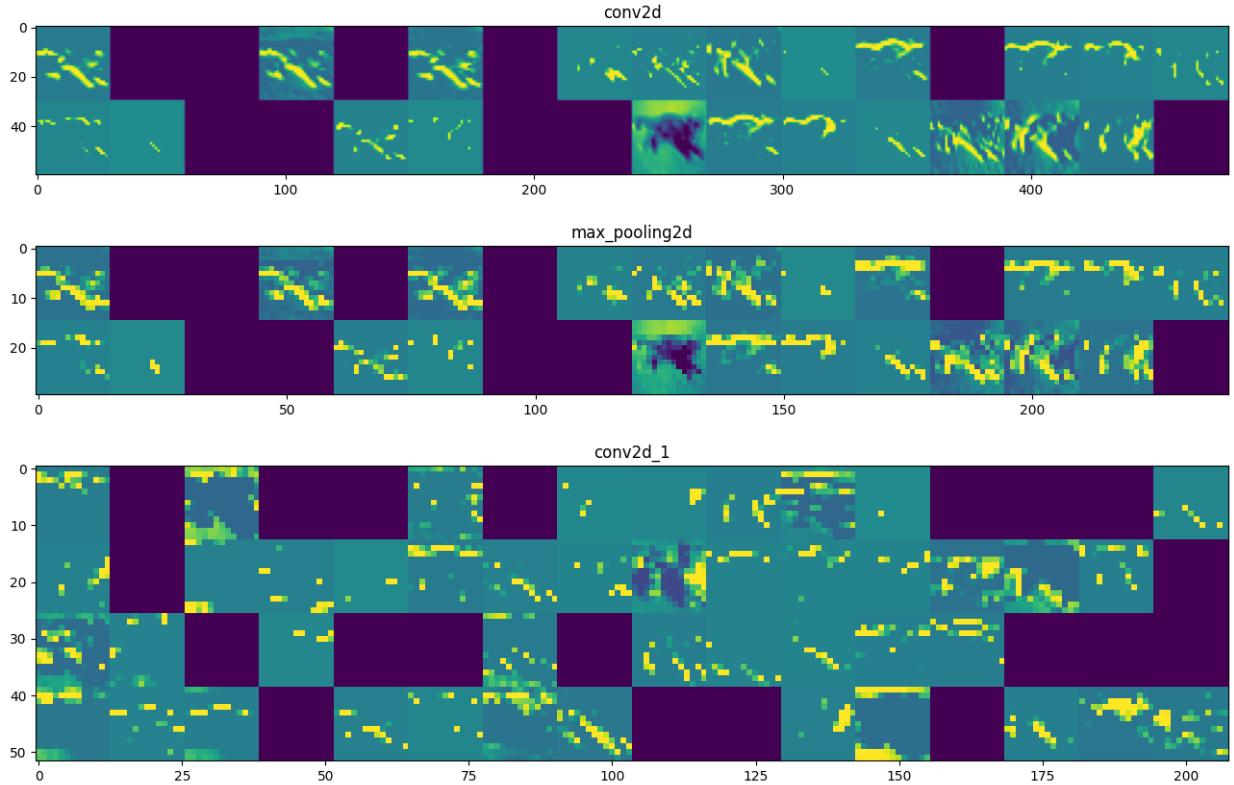
plt.show();
```



```
1/1 [=====] - 0s 74ms/step
```

```
<ipython-input-164-3bac8bdd9965>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[159]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):

```

```

channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

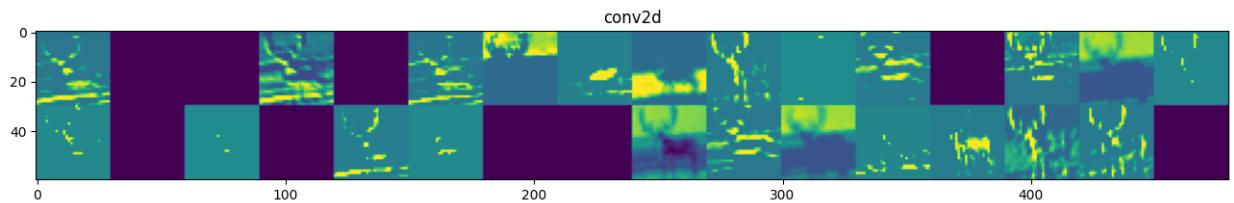
```

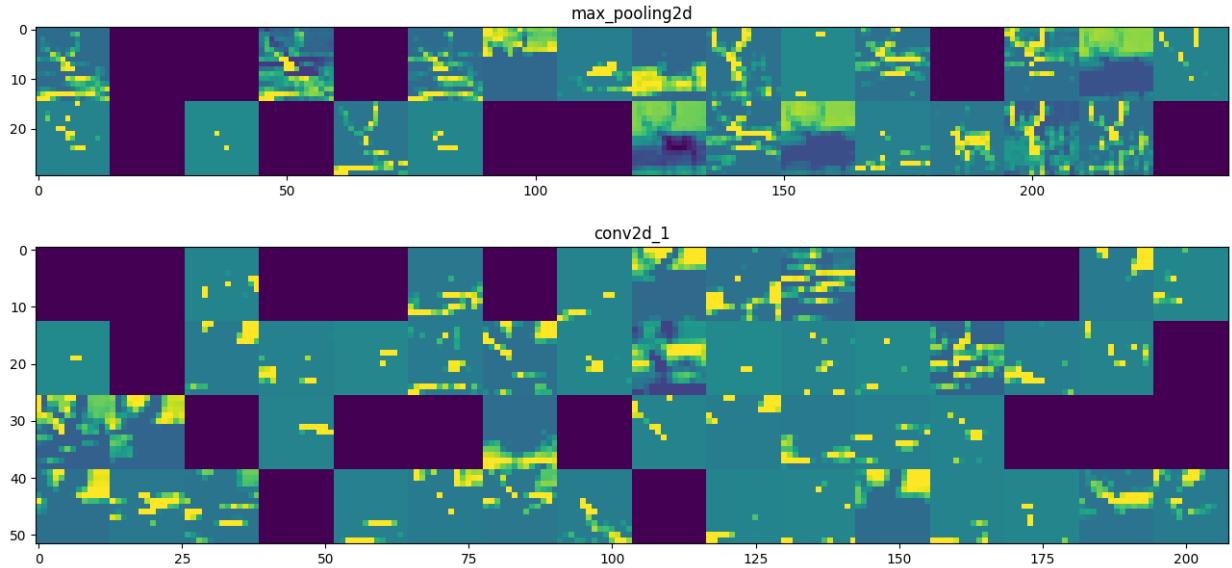


1/1 [=====] - 0s 83ms/step

<ipython-input-165-a52112e96c59>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[24]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

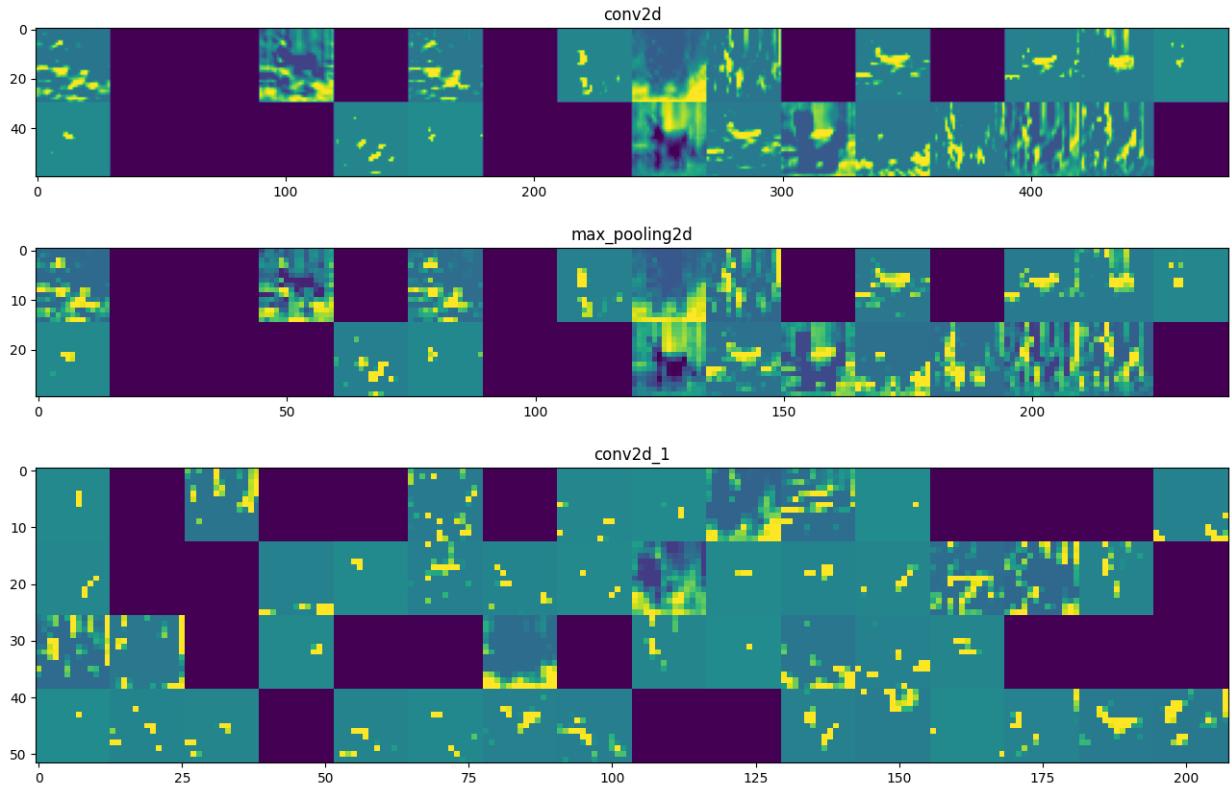
plt.show();
```



```
1/1 [=====] - 0s 88ms/step
```

```
<ipython-input-166-edee34e9d17b>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[152]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

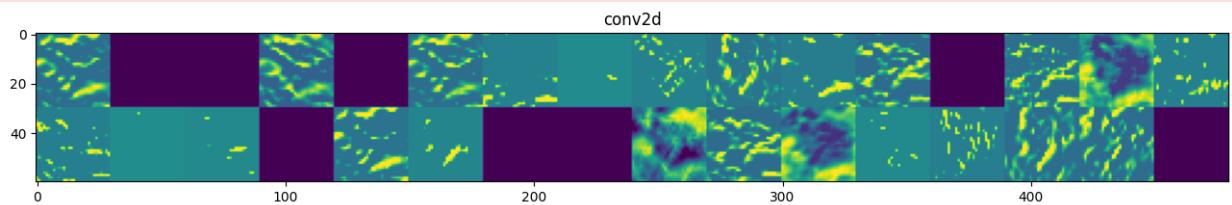
```

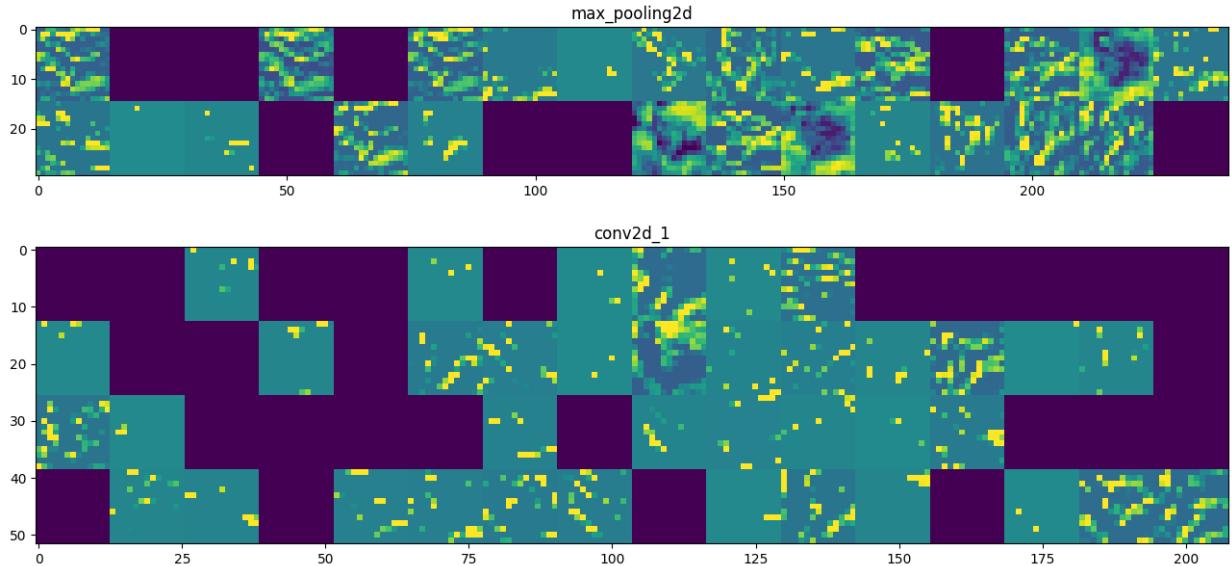


1/1 [=====] - 0s 78ms/step

<ipython-input-167-0ef3f9c61bfb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[2004]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

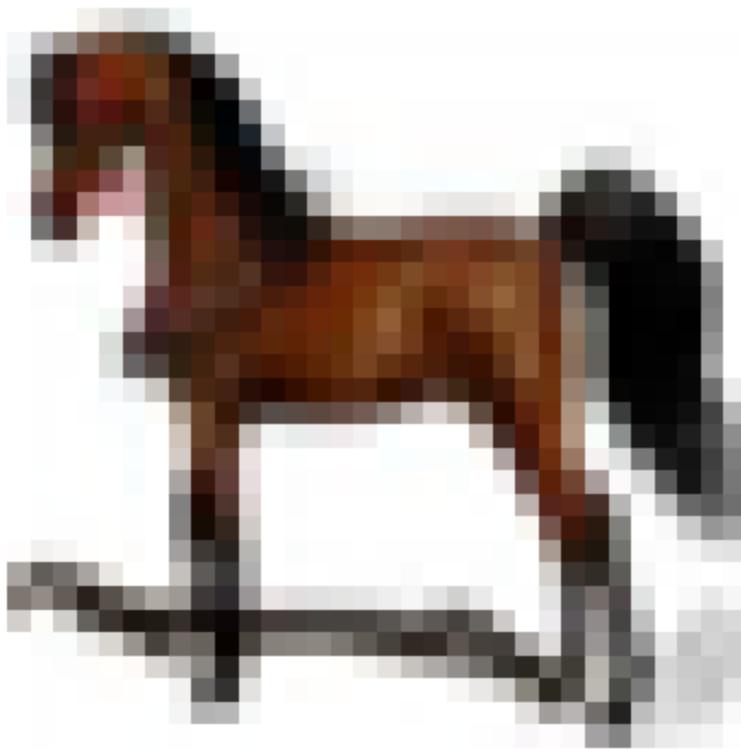
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

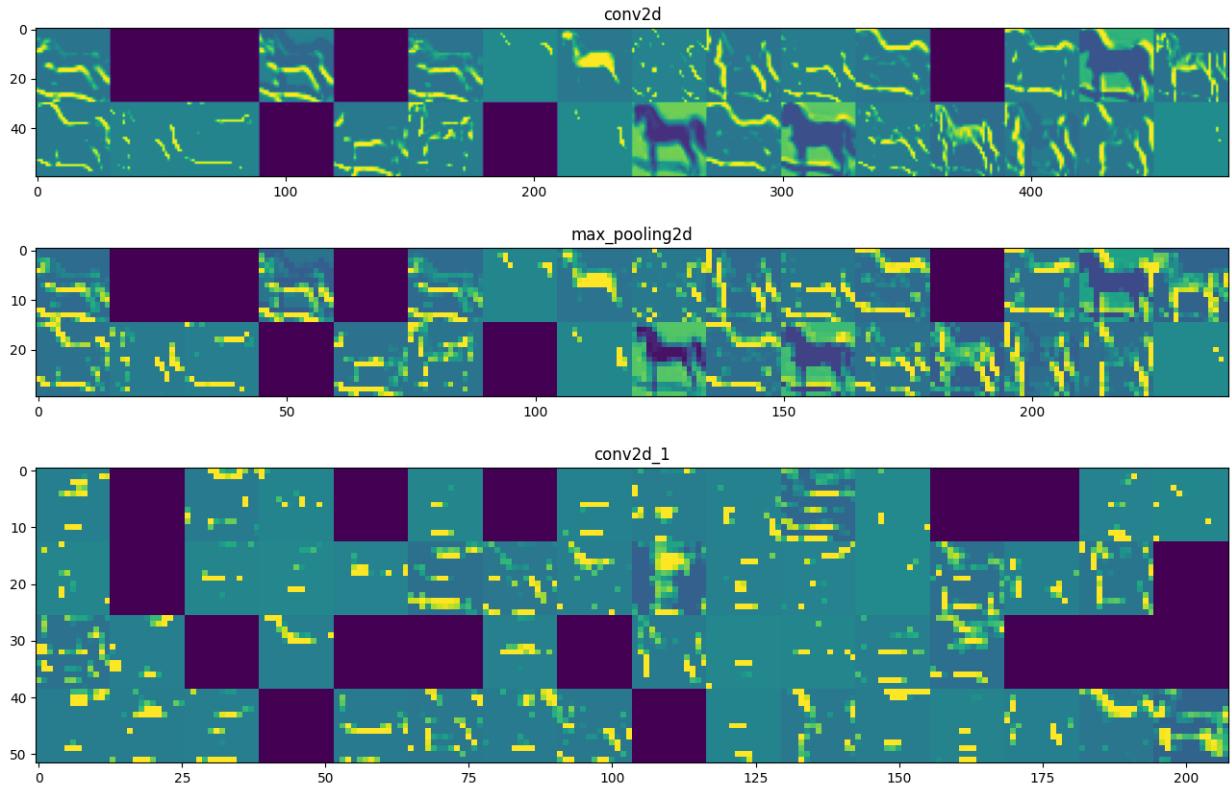
plt.show();
```



```
1/1 [=====] - 0s 116ms/step
```

```
<ipython-input-168-5c55fce06c66>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[185]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

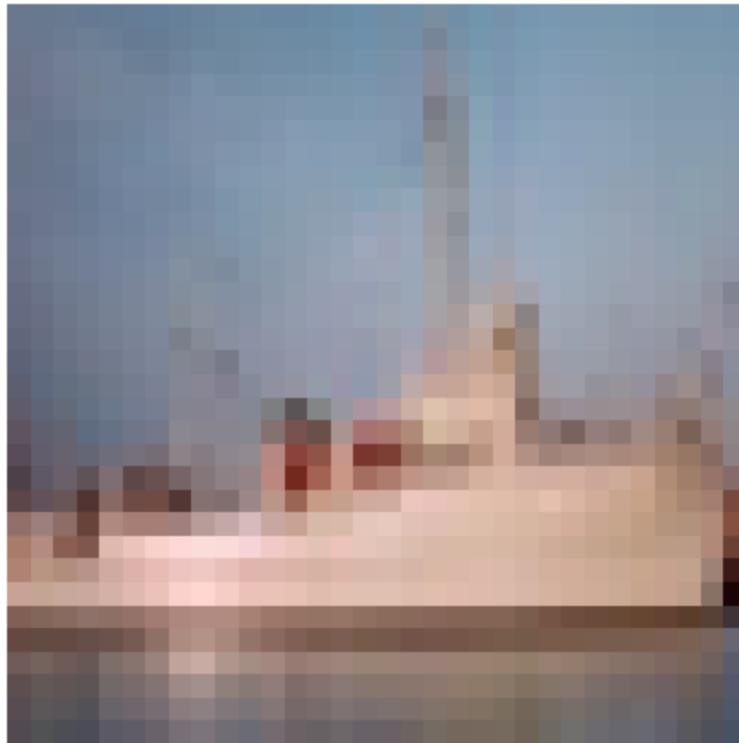
channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

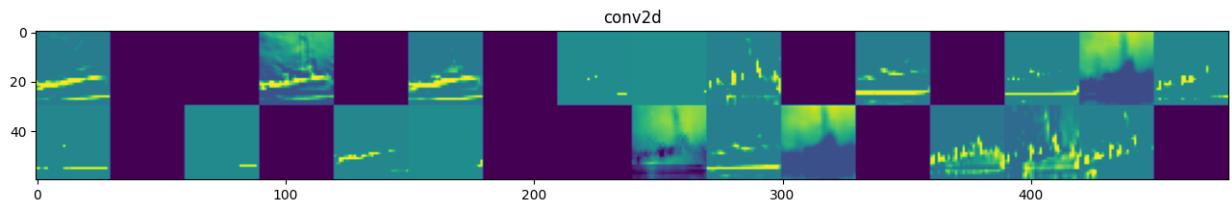
```

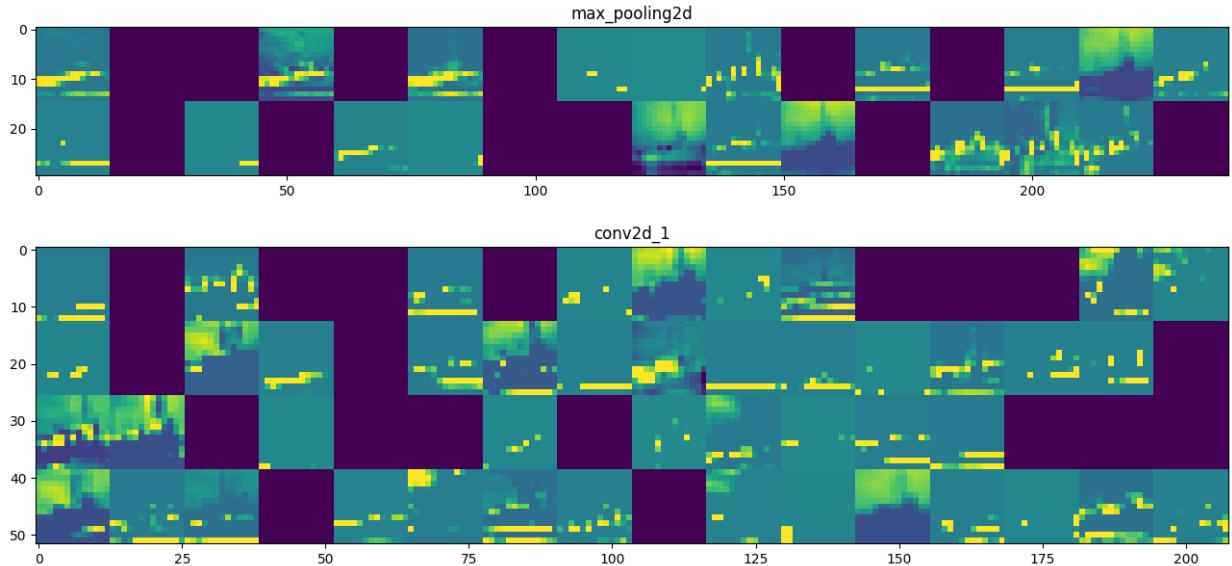


1/1 [=====] - 0s 81ms/step

<ipython-input-169-f532589aadeb>:72: RuntimeWarning: invalid value encountered in divide

```
channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[133]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

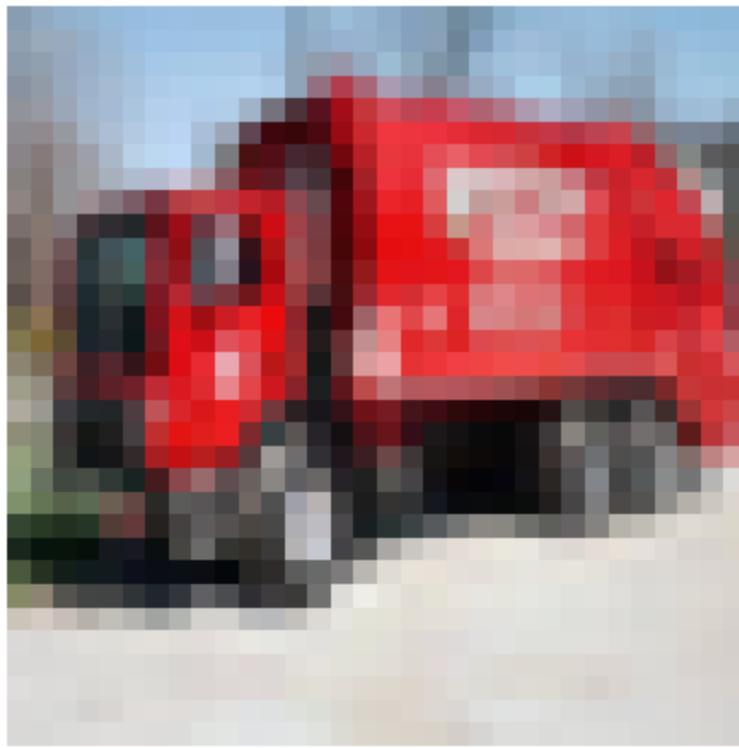
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

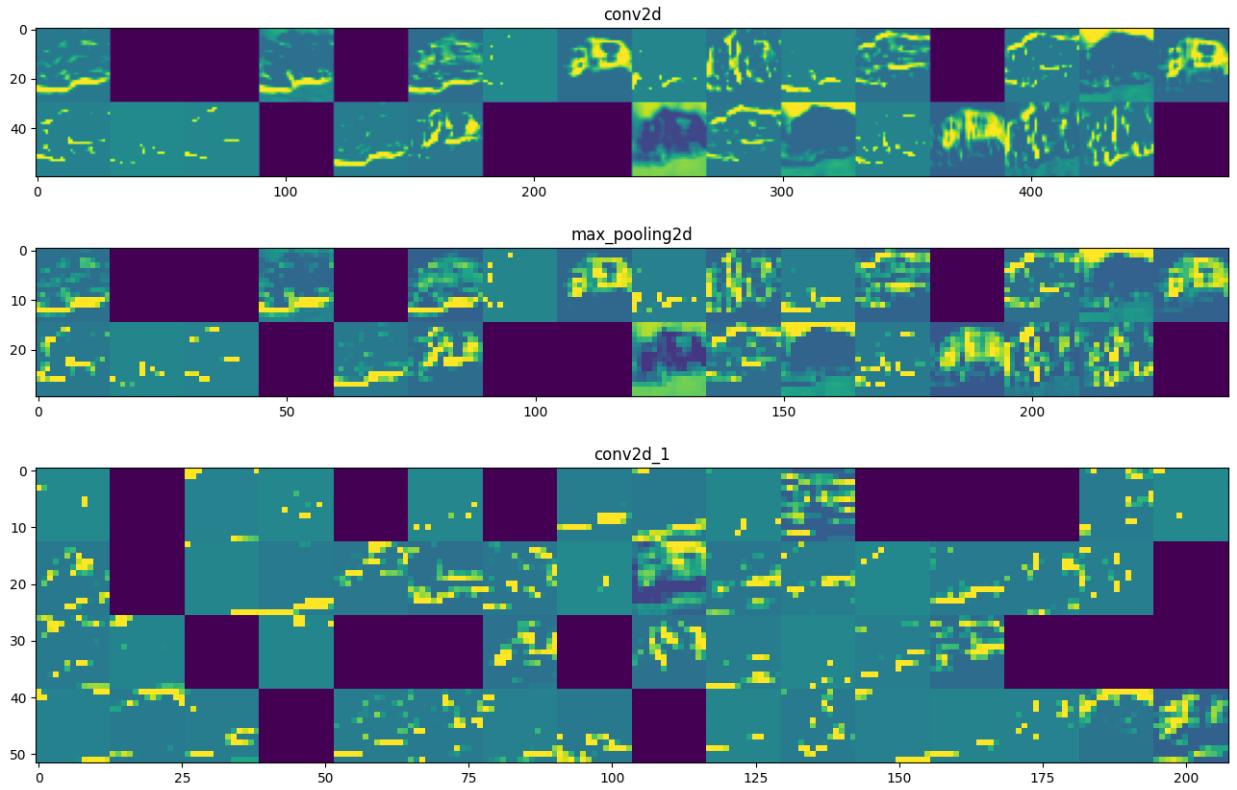
plt.show();
```



```
1/1 [=====] - 0s 79ms/step
```

```
<ipython-input-170-8e6847644768>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



6) Model 5 - CNN with 3 Max Pooling / Hidden Layers and Early Stopping Regularization

6.1) Build The Model

We use a Sequential class defined in Keras to create our model.

```
In [ ]: k.clear_session()
model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu),
    MaxPool2D((2, 2),strides=2),
    Flatten(),
    Dense(units=256,activation=tf.nn.softmax),
    Dense(units=10, activation=tf.nn.softmax)
])
```

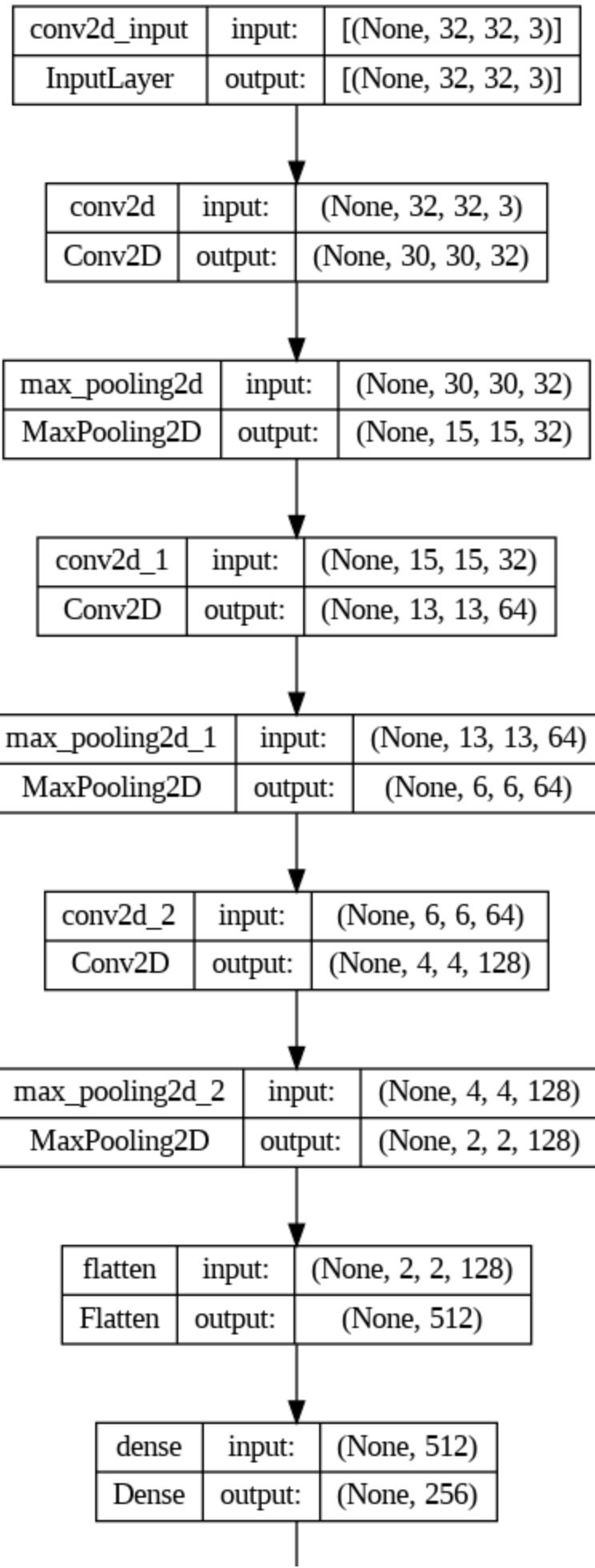
```
In [ ]: model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 256)	131328
dense_1 (Dense)	(None, 10)	2570
<hr/>		
Total params: 227146 (887.29 KB)		
Trainable params: 227146 (887.29 KB)		
Non-trainable params: 0 (0.00 Byte)		

```
In [ ]: tf.keras.utils.plot_model(model, "CIFAR10.png", show_shapes=True)
```

Out[]:



dense_1	input:	(None, 256)
Dense	output:	(None, 10)

Let's now compile and train the model.

tf.keras.losses.SparseCategoricalCrossentropy

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Module: tf.keras.callbacks

tf.keras.callbacks.EarlyStopping

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

tf.keras.callbacks.ModelCheckpoint

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

```
In [ ]: model.compile(optimizer='adam',
                      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
                      metrics=['accuracy'])
```

```
In [ ]: history = model.fit(x_train_norm
                           ,y_train_split
                           ,epochs=30
                           ,validation_data=(x_valid_norm, y_valid_split)
                           ,callbacks=[
                           tf.keras.callbacks.ModelCheckpoint("Model_5", save_best_only=True,
                           ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=5)
                           ])
```

```

Epoch 1/30
1407/1407 [=====] - 41s 29ms/step - loss: 2.0905 - accuracy: 0.2117 - val_loss: 1.9452 - val_accuracy: 0.2408
Epoch 2/30
1407/1407 [=====] - 39s 28ms/step - loss: 1.8556 - accuracy: 0.2902 - val_loss: 1.7440 - val_accuracy: 0.3760
Epoch 3/30
1407/1407 [=====] - 39s 28ms/step - loss: 1.6514 - accuracy: 0.4055 - val_loss: 1.5725 - val_accuracy: 0.4348
Epoch 4/30
1407/1407 [=====] - 40s 28ms/step - loss: 1.4613 - accuracy: 0.4854 - val_loss: 1.4827 - val_accuracy: 0.4632
Epoch 5/30
1407/1407 [=====] - 44s 32ms/step - loss: 1.3158 - accuracy: 0.5428 - val_loss: 1.2775 - val_accuracy: 0.5538
Epoch 6/30
1407/1407 [=====] - 39s 28ms/step - loss: 1.1855 - accuracy: 0.5891 - val_loss: 1.2177 - val_accuracy: 0.5706
Epoch 7/30
1407/1407 [=====] - 39s 28ms/step - loss: 1.0866 - accuracy: 0.6198 - val_loss: 1.1539 - val_accuracy: 0.6010
Epoch 8/30
1407/1407 [=====] - 41s 29ms/step - loss: 1.0158 - accuracy: 0.6427 - val_loss: 1.1117 - val_accuracy: 0.6044
Epoch 9/30
1407/1407 [=====] - 39s 28ms/step - loss: 0.9536 - accuracy: 0.6641 - val_loss: 1.0745 - val_accuracy: 0.6168
Epoch 10/30
1407/1407 [=====] - 39s 28ms/step - loss: 0.8995 - accuracy: 0.6808 - val_loss: 1.0721 - val_accuracy: 0.6218
Epoch 11/30
1407/1407 [=====] - 39s 28ms/step - loss: 0.8606 - accuracy: 0.6957 - val_loss: 1.0526 - val_accuracy: 0.6312
Epoch 12/30
1407/1407 [=====] - 39s 28ms/step - loss: 0.8181 - accuracy: 0.7088 - val_loss: 1.0520 - val_accuracy: 0.6356
Epoch 13/30
1407/1407 [=====] - 38s 27ms/step - loss: 0.7744 - accuracy: 0.7228 - val_loss: 1.0750 - val_accuracy: 0.6278
Epoch 14/30
1407/1407 [=====] - 38s 27ms/step - loss: 0.7487 - accuracy: 0.7313 - val_loss: 1.1062 - val_accuracy: 0.6228
Epoch 15/30
1407/1407 [=====] - 38s 27ms/step - loss: 0.7223 - accuracy: 0.7413 - val_loss: 1.0983 - val_accuracy: 0.6280

```

6.2) Evaluate Model Performance

```
In [ ]: model = tf.keras.models.load_model("Model_5")
print(f"Training accuracy: {model.evaluate(x_train_norm, y_train_split)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_norm, y_valid_split)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_norm, y_test)[1]:.3f}")
```

```
1407/1407 [=====] - 17s 12ms/step - loss: 0.7439 - accuracy: 0.7392
Training accuracy: 0.739
157/157 [=====] - 2s 12ms/step - loss: 1.0520 - accuracy: 0.6356
Validation accuracy: 0.636
313/313 [=====] - 3s 8ms/step - loss: 1.0713 - accuracy: 0.6306
Test accuracy: 0.631
```

```
In [ ]: preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

```
313/313 [=====] - 3s 8ms/step
shape of preds: (10000, 10)
```

```
In [ ]: history_dict = history.history
history_dict.keys()
```

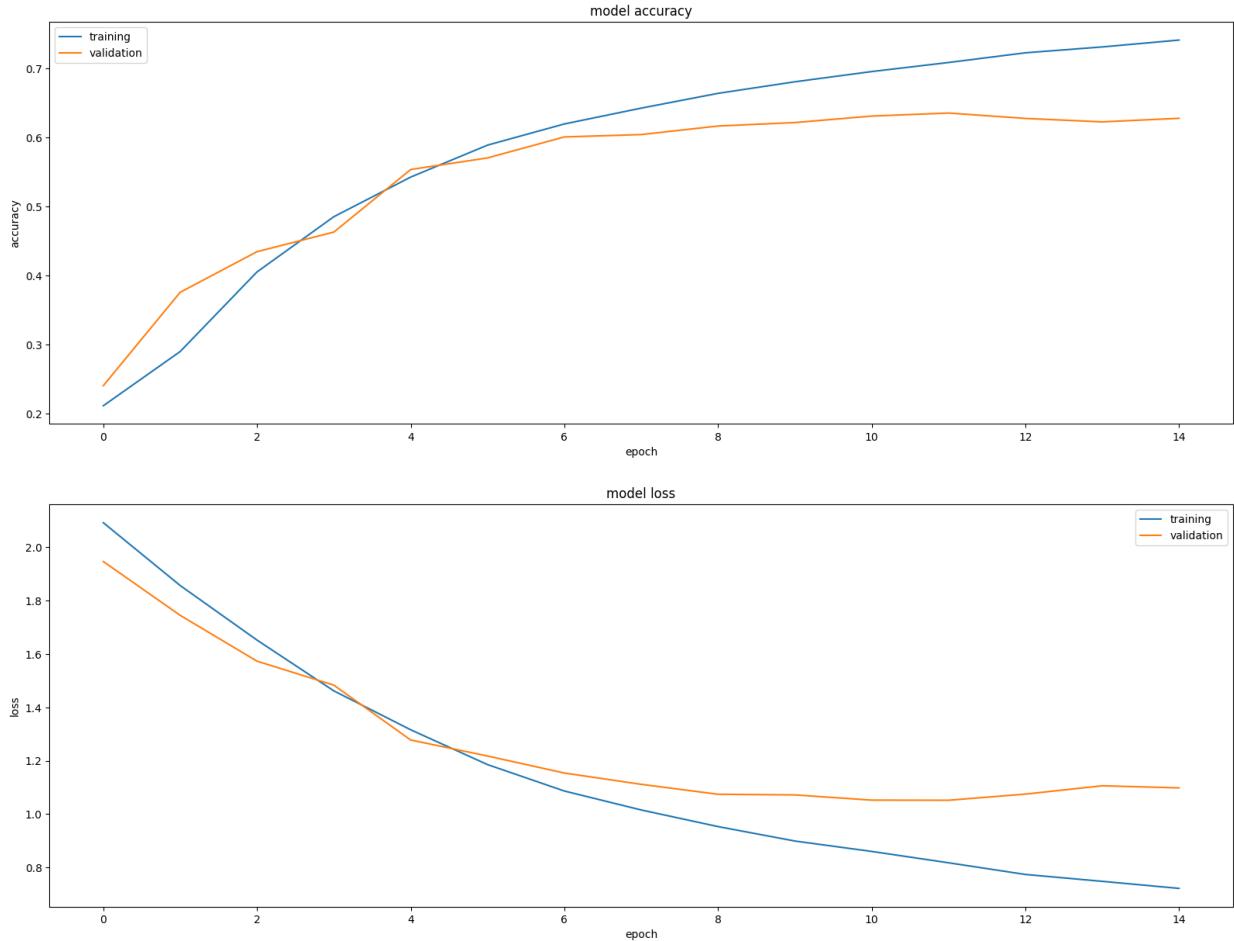
```
Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)
```

	loss	accuracy	val_loss	val_accuracy
10	0.861	0.696	1.053	0.631
11	0.818	0.709	1.052	0.636
12	0.774	0.723	1.075	0.628
13	0.749	0.731	1.106	0.623
14	0.722	0.741	1.098	0.628

```
In [ ]: plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```

```
<ipython-input-8-353fbbe40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.
    ax = plt.subplot(subplot)
```



Let's examine the precision, recall, F1 score, and confusion matrix.

```
In [ ]: pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

313/313 [=====] - 3s 11ms/step
```

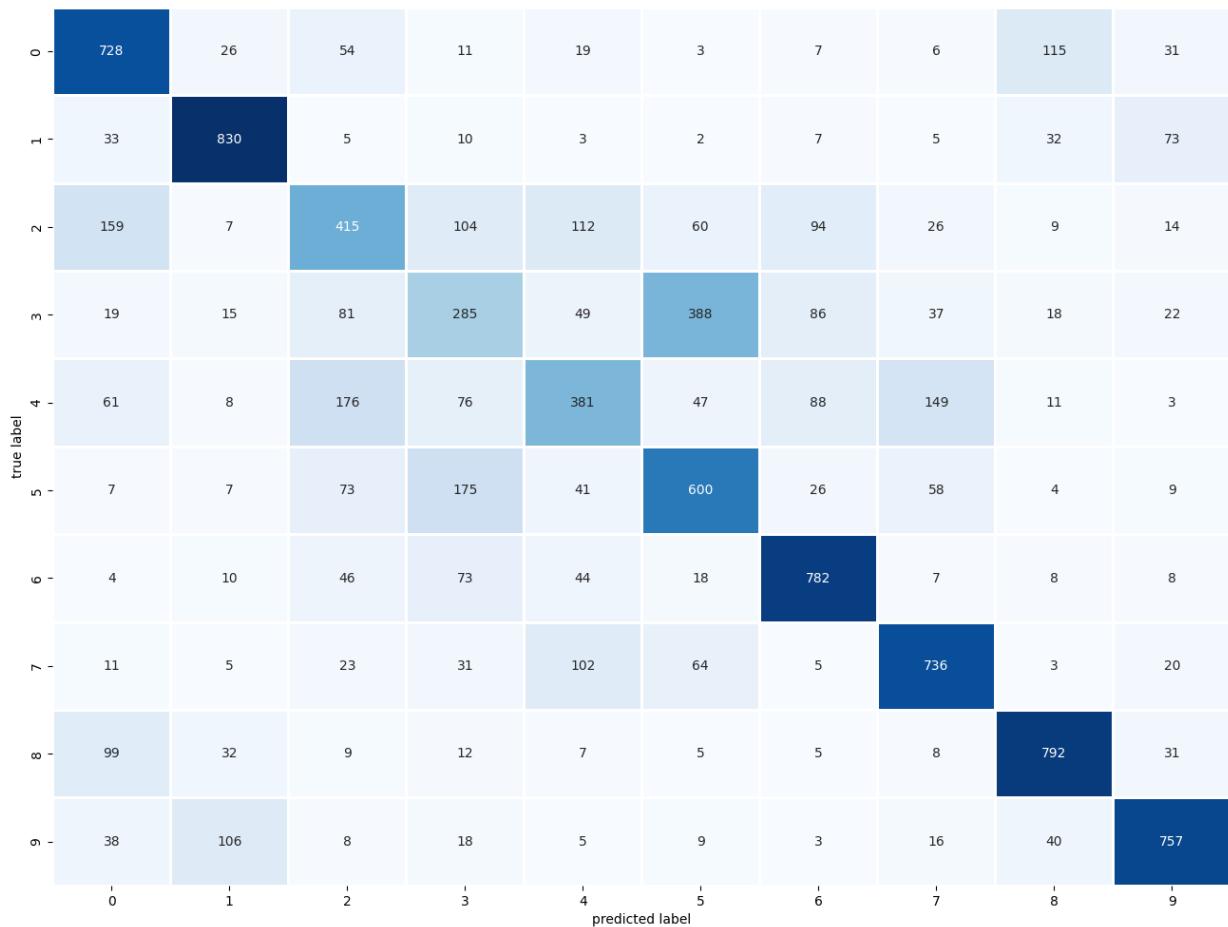
```
In [ ]: print_validation_report(y_test, pred1)
```

	precision	recall	f1-score	support
0	0.63	0.73	0.67	1000
1	0.79	0.83	0.81	1000
2	0.47	0.41	0.44	1000
3	0.36	0.28	0.32	1000
4	0.50	0.38	0.43	1000
5	0.50	0.60	0.55	1000
6	0.71	0.78	0.74	1000
7	0.70	0.74	0.72	1000
8	0.77	0.79	0.78	1000
9	0.78	0.76	0.77	1000
accuracy			0.63	10000
macro avg	0.62	0.63	0.62	10000
weighted avg	0.62	0.63	0.62	10000

Accuracy Score: 0.6306

Root Mean Square Error: 2.4170436487577134

In []: `plot_confusion_matrix(y_test,pred1)`



Load HDF5 Model Format

`tf.keras.models.load_model`

https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model

In []: `model = tf.keras.models.load_model('Model_5')
preds = model.predict(x_test_norm)
preds.shape`

313/313 [=====] - 2s 8ms/step

Out[]: `(10000, 10)`

Let's examine the predictions for the testing dataset.

In []: `cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)

df = pd.DataFrame(preds[0:20], columns = ['airplane'
,'automobile'
, 'bird'
, 'cat'
, 'deer'
, 'dog'
, 'frog'
, 'horse'`

```

        , 'ship'
        , 'truck'])
df.style.format("{:.2%}").background_gradient(cmap=cm)

```

Out[]:

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0	0.03%	0.06%	4.31%	39.72%	1.26%	52.20%	0.90%	1.25%	0.17%	0.11%
1	11.17%	49.90%	0.30%	1.42%	0.17%	0.35%	0.47%	0.40%	28.74%	7.09%
2	8.63%	7.68%	0.06%	0.53%	0.04%	0.09%	0.14%	0.14%	79.49%	3.21%
3	80.63%	0.10%	11.48%	0.64%	3.97%	0.12%	0.15%	0.11%	2.54%	0.25%
4	0.02%	0.15%	1.62%	1.47%	3.44%	0.14%	92.97%	0.03%	0.07%	0.10%
5	0.04%	0.34%	3.64%	4.79%	5.67%	0.67%	84.30%	0.10%	0.18%	0.27%
6	0.25%	97.80%	0.10%	0.27%	0.06%	0.15%	0.15%	0.11%	0.09%	1.01%
7	0.05%	0.25%	3.41%	2.52%	7.18%	0.29%	85.90%	0.07%	0.14%	0.20%
8	0.10%	0.19%	8.76%	36.07%	6.94%	39.87%	2.87%	4.52%	0.33%	0.35%
9	0.45%	96.30%	0.14%	0.38%	0.09%	0.19%	0.20%	0.16%	0.16%	1.92%
10	38.63%	0.18%	33.60%	3.10%	19.41%	1.14%	0.51%	1.05%	1.79%	0.58%
11	0.63%	0.90%	0.09%	0.33%	0.03%	0.06%	0.11%	0.19%	0.22%	97.44%
12	0.11%	0.25%	9.10%	35.89%	8.51%	36.21%	4.18%	4.95%	0.37%	0.43%
13	0.03%	0.05%	0.18%	0.20%	8.67%	0.77%	0.03%	89.88%	0.02%	0.16%
14	0.78%	1.15%	0.11%	0.40%	0.04%	0.08%	0.14%	0.23%	0.29%	96.78%
15	3.73%	8.26%	6.84%	9.87%	7.40%	1.31%	47.08%	0.57%	11.03%	3.93%
16	0.01%	0.04%	3.16%	37.44%	0.90%	56.55%	0.69%	1.05%	0.10%	0.07%
17	0.98%	0.82%	12.16%	25.70%	13.99%	23.98%	2.67%	15.44%	1.01%	3.24%
18	13.20%	6.82%	0.13%	0.79%	0.07%	0.13%	0.20%	0.21%	71.78%	6.68%
19	0.05%	0.39%	3.76%	3.86%	10.27%	0.63%	80.33%	0.22%	0.18%	0.31%

In []:

```

(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[98]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
    , 'automobile'
    , 'bird'
    , 'cat'
    , 'deer'
    , 'dog'
    , 'frog'
    , 'horse'
    , 'ship'
    , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')

```

```
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

# Display the grid
```

```
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                    scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

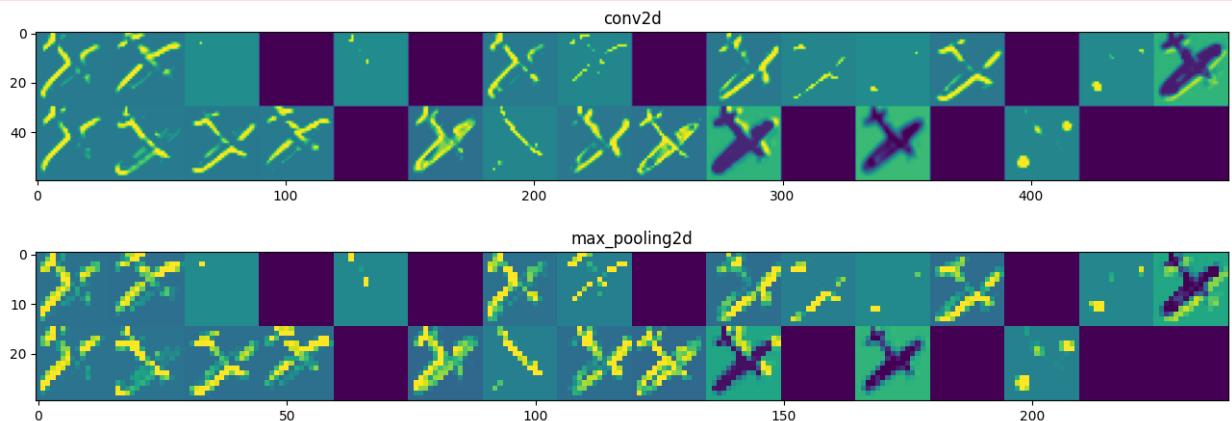
plt.show();
```

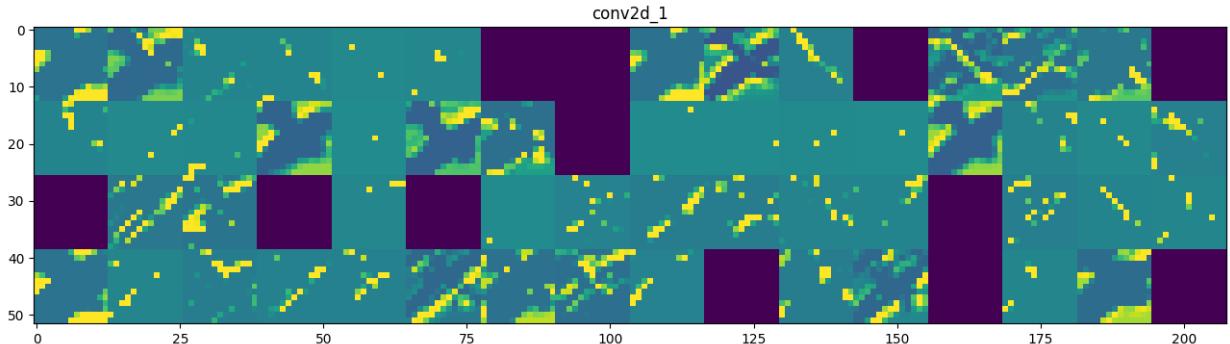


1/1 [=====] - 0s 80ms/step

<ipython-input-211-9ab60ebbedca>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[122]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)
```

```
images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

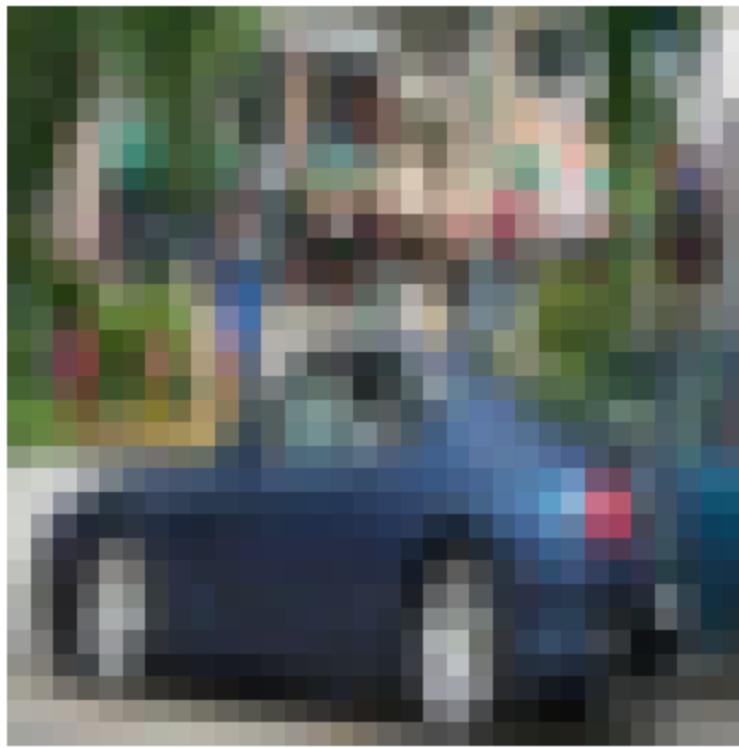
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

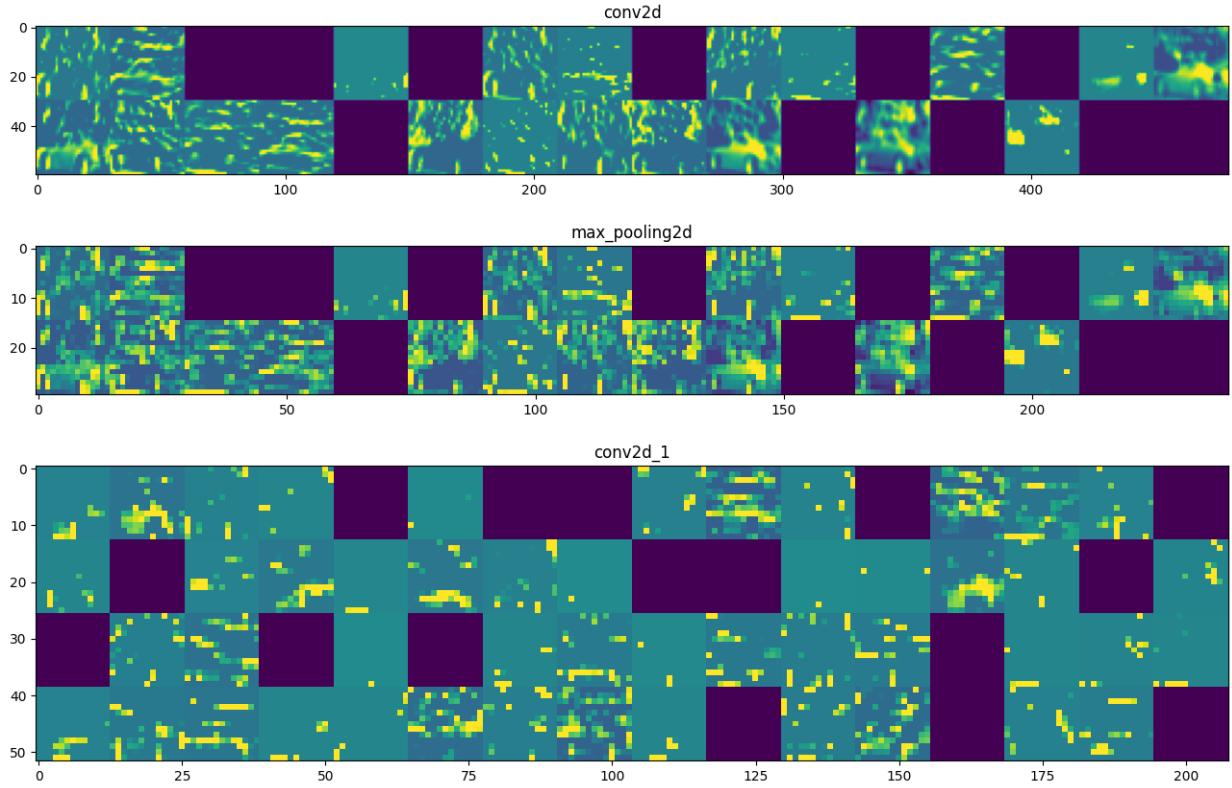
plt.show();
```



```
1/1 [=====] - 0s 87ms/step
```

```
<ipython-input-212-e0d043c5b9b7>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[75]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

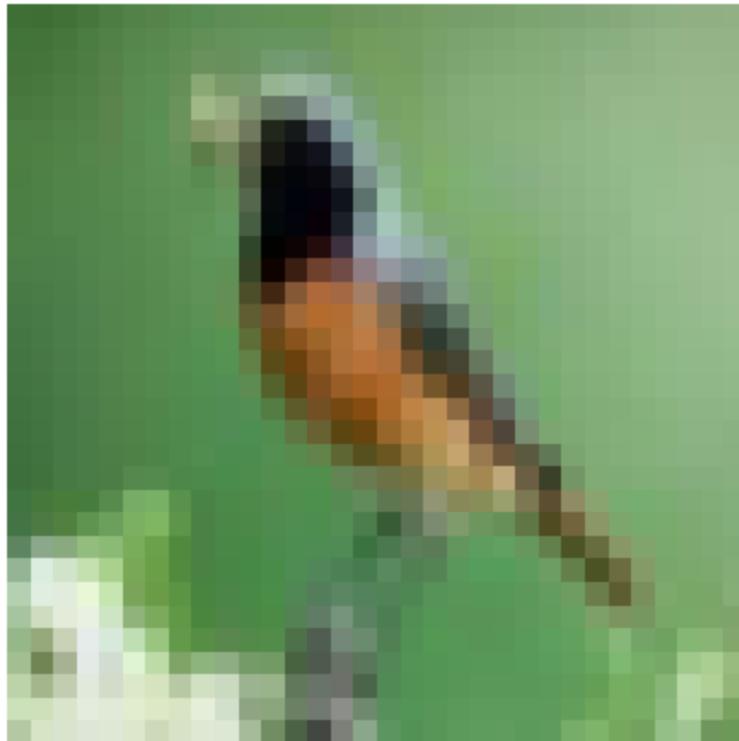
channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

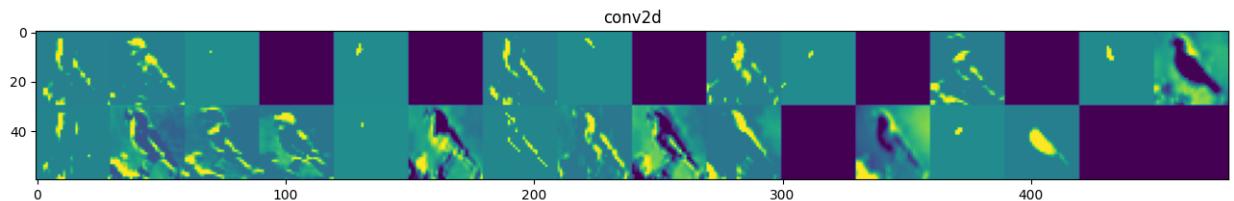
```

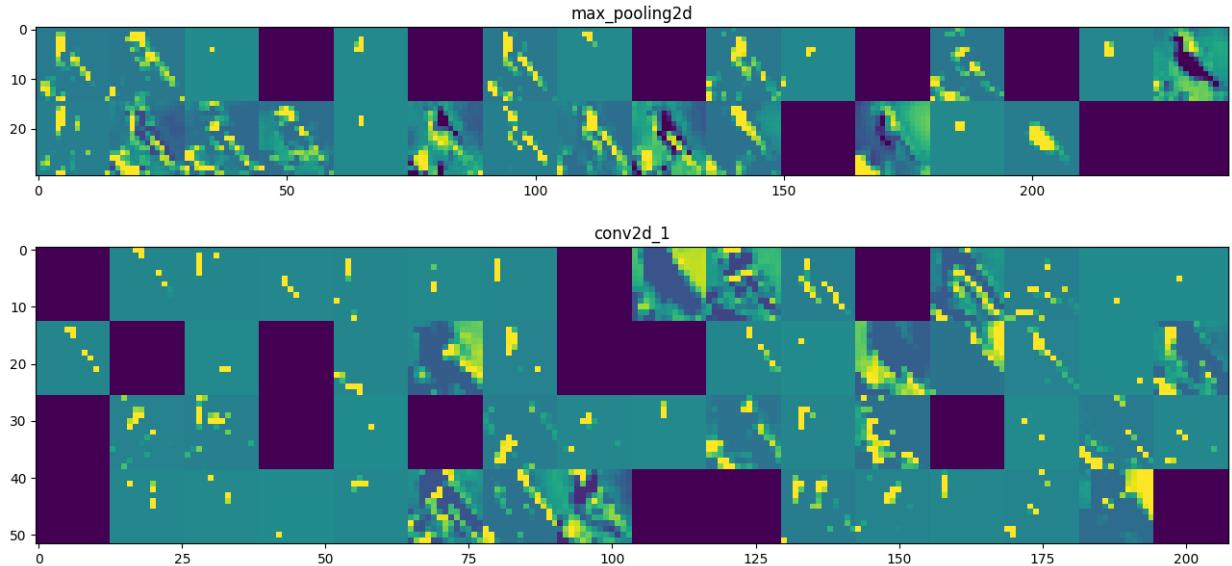


1/1 [=====] - 0s 75ms/step

<ipython-input-213-4848b71b261c>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[184]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

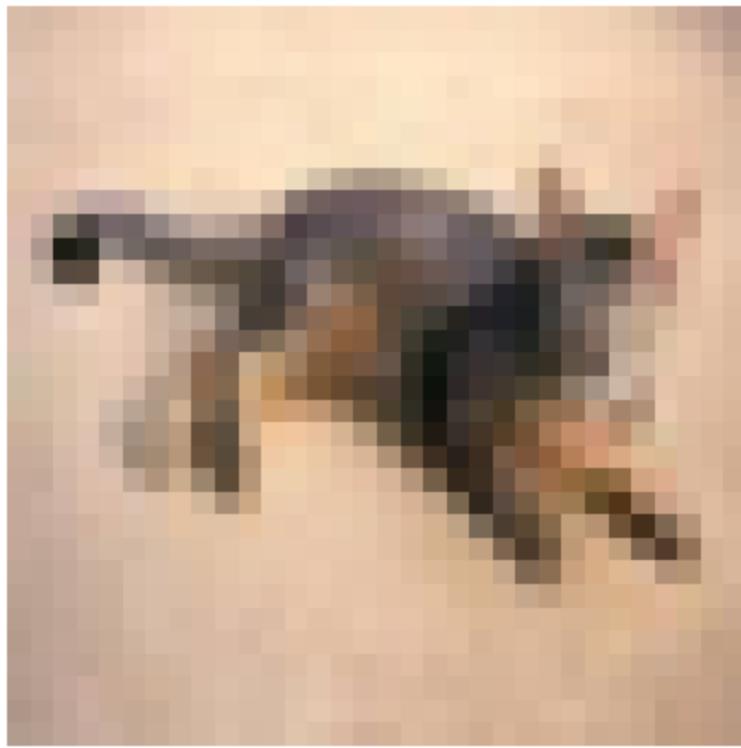
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

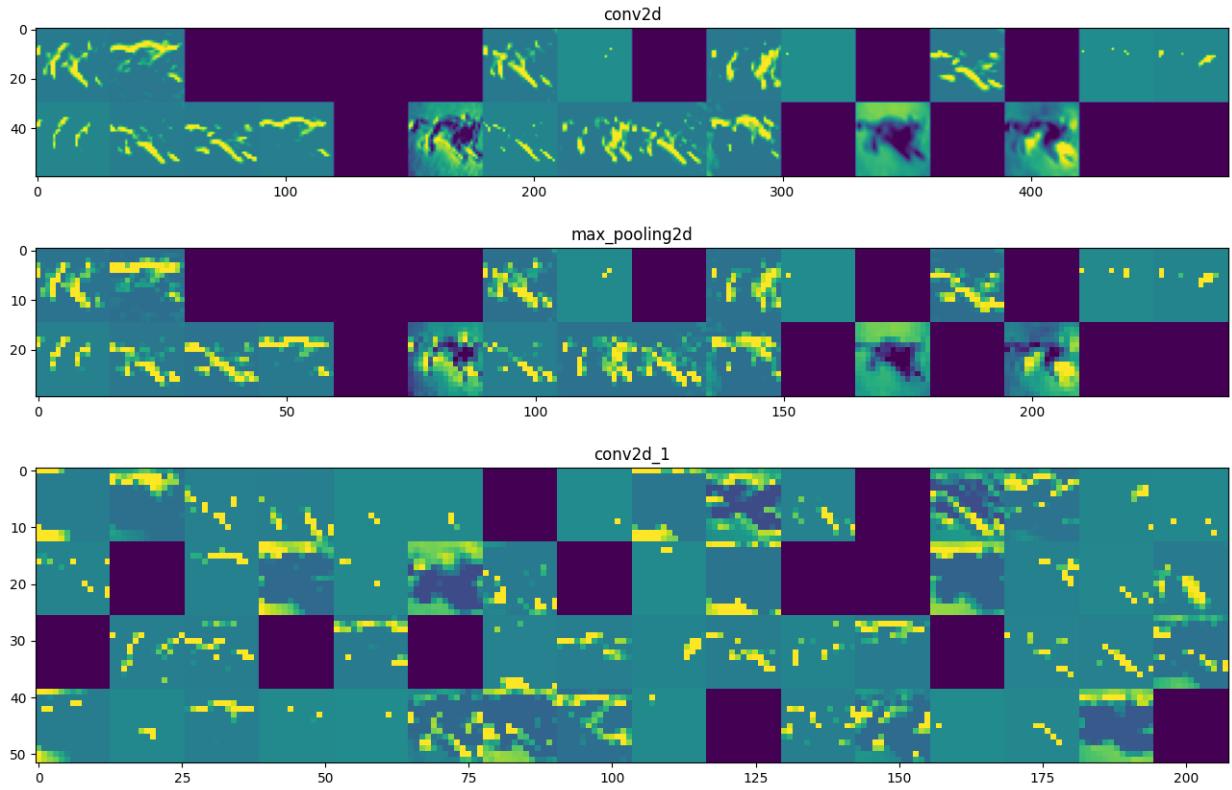
plt.show();
```



```
1/1 [=====] - 0s 79ms/step
```

```
<ipython-input-214-3bac8bdd9965>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[159]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

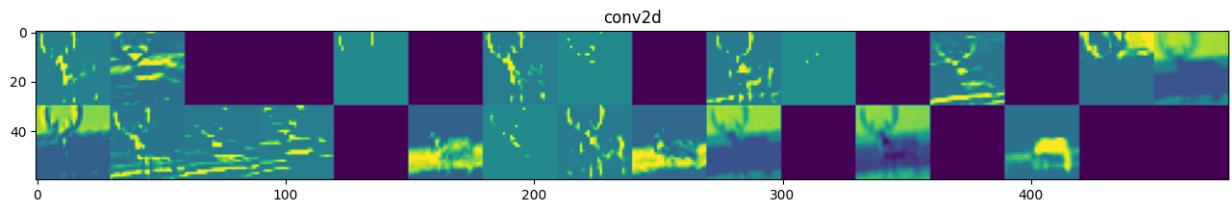
```

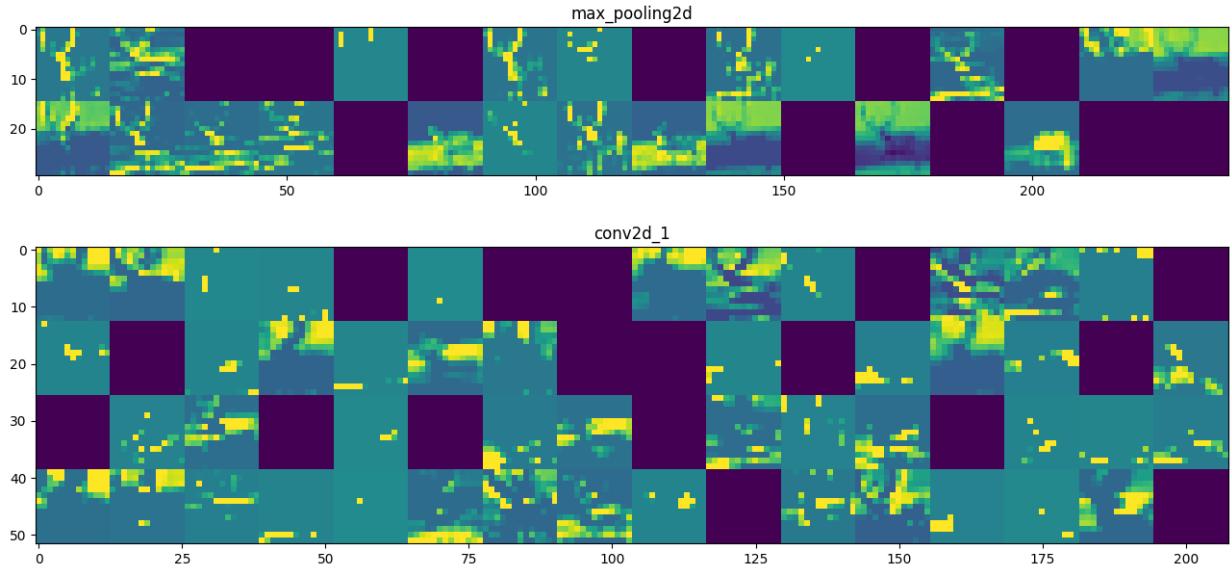


1/1 [=====] - 0s 80ms/step

<ipython-input-215-a52112e96c59>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[24]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

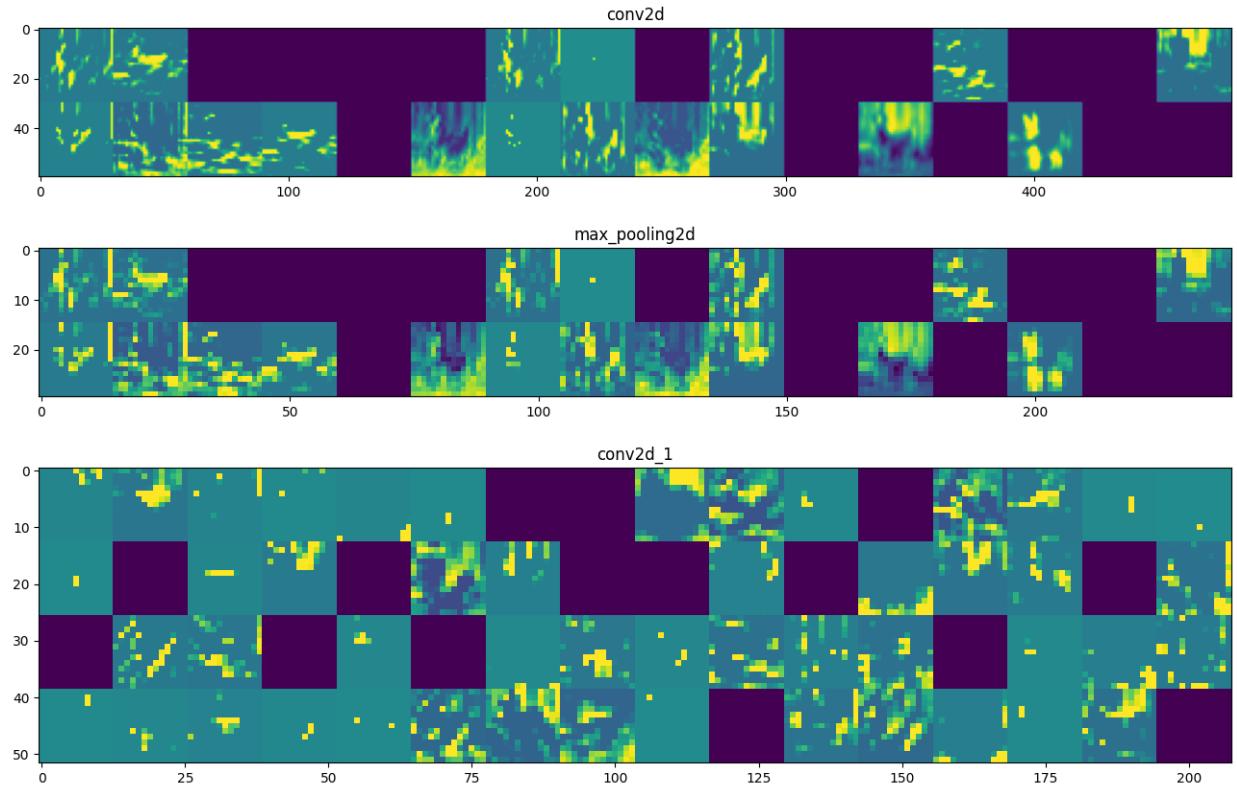
plt.show();
```



```
1/1 [=====] - 0s 86ms/step
```

```
<ipython-input-216-edee34e9d17b>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[152]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

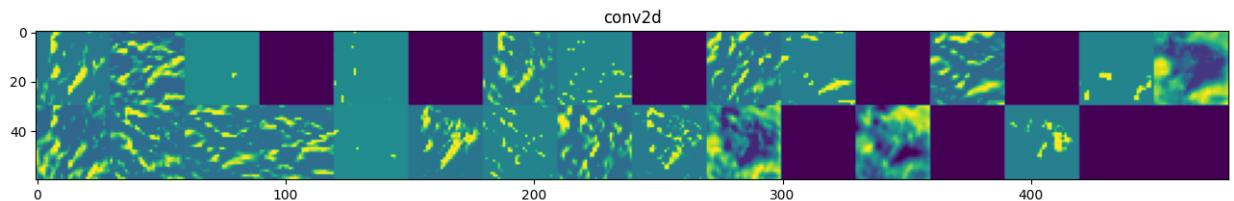
```

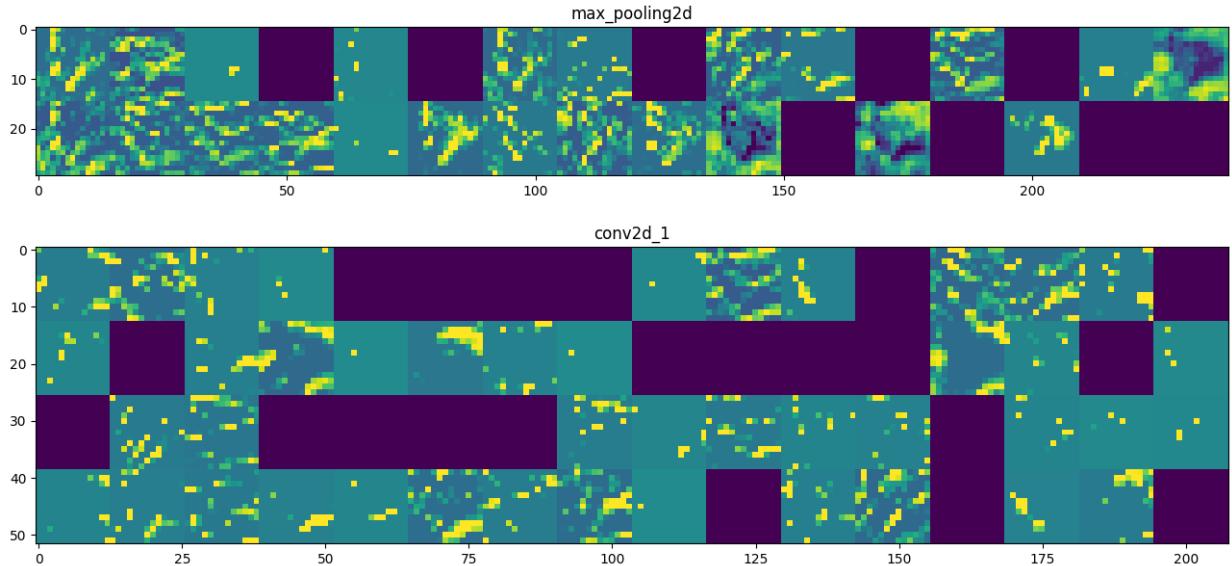


1/1 [=====] - 0s 77ms/step

<ipython-input-217-0ef3f9c61bfb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[2004]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

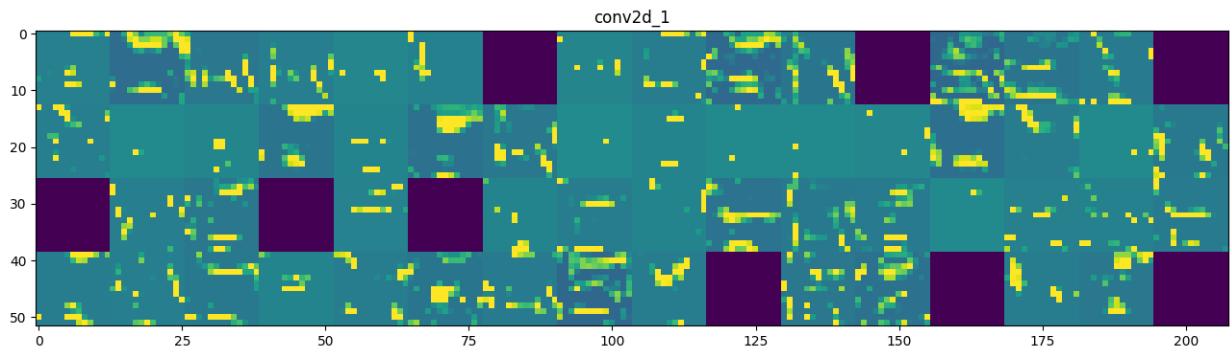
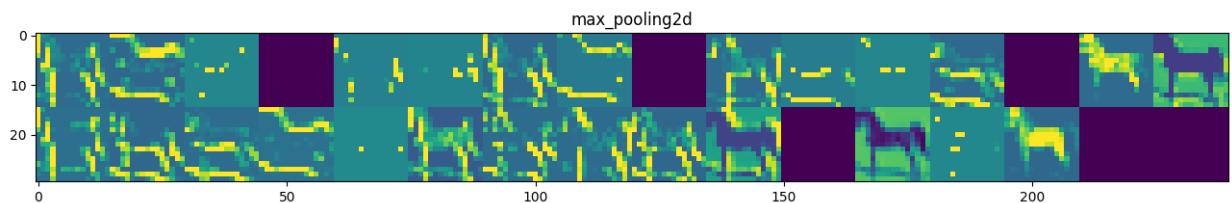
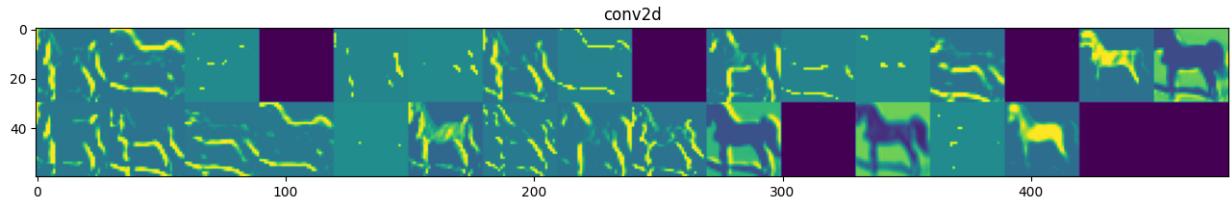
plt.show();
```



```
1/1 [=====] - 0s 84ms/step
```

```
<ipython-input-218-5c55fce06c66>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[185]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

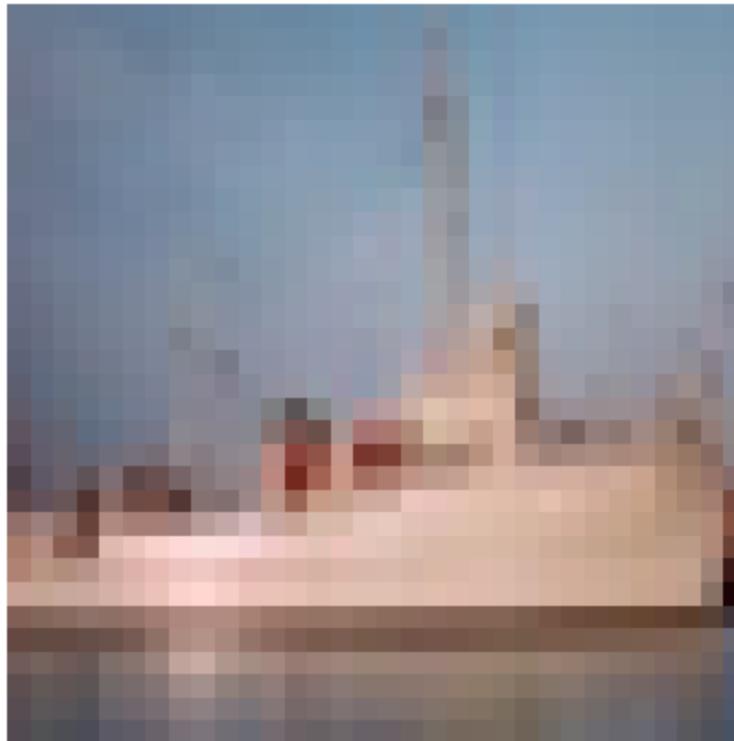
channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

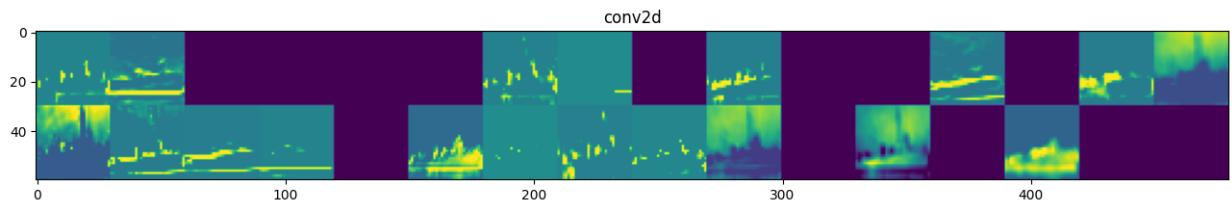
```

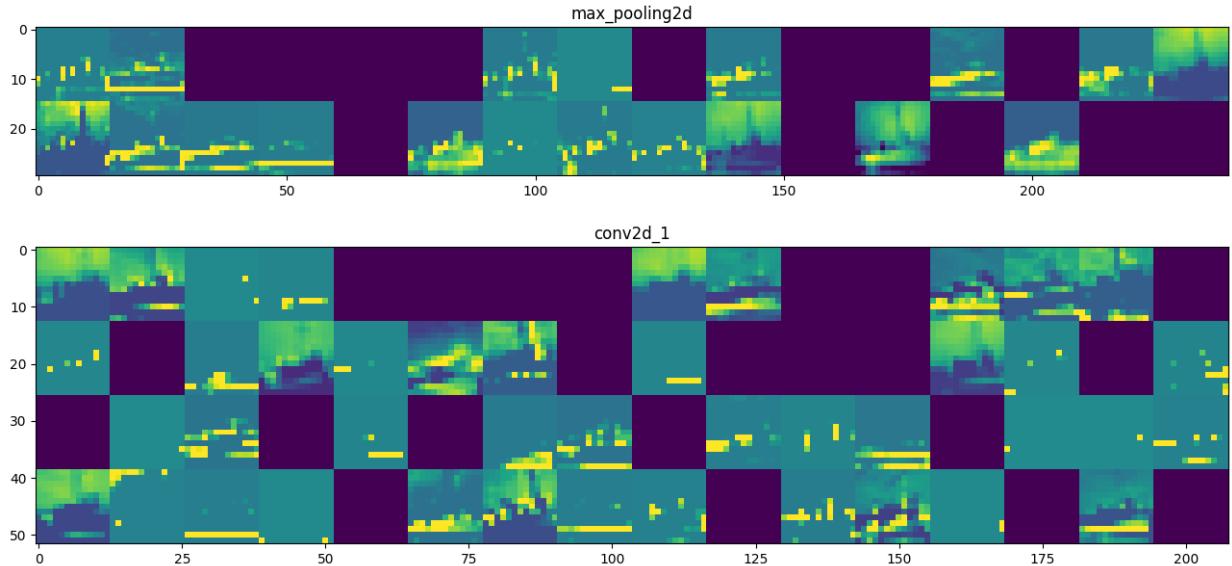


1/1 [=====] - 0s 84ms/step

<ipython-input-219-f532589aadeb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[133]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

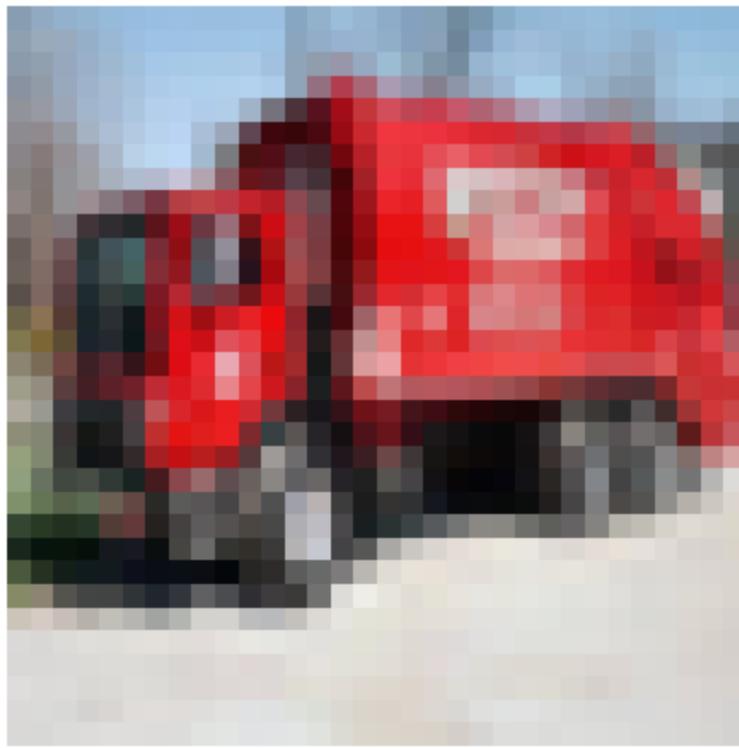
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

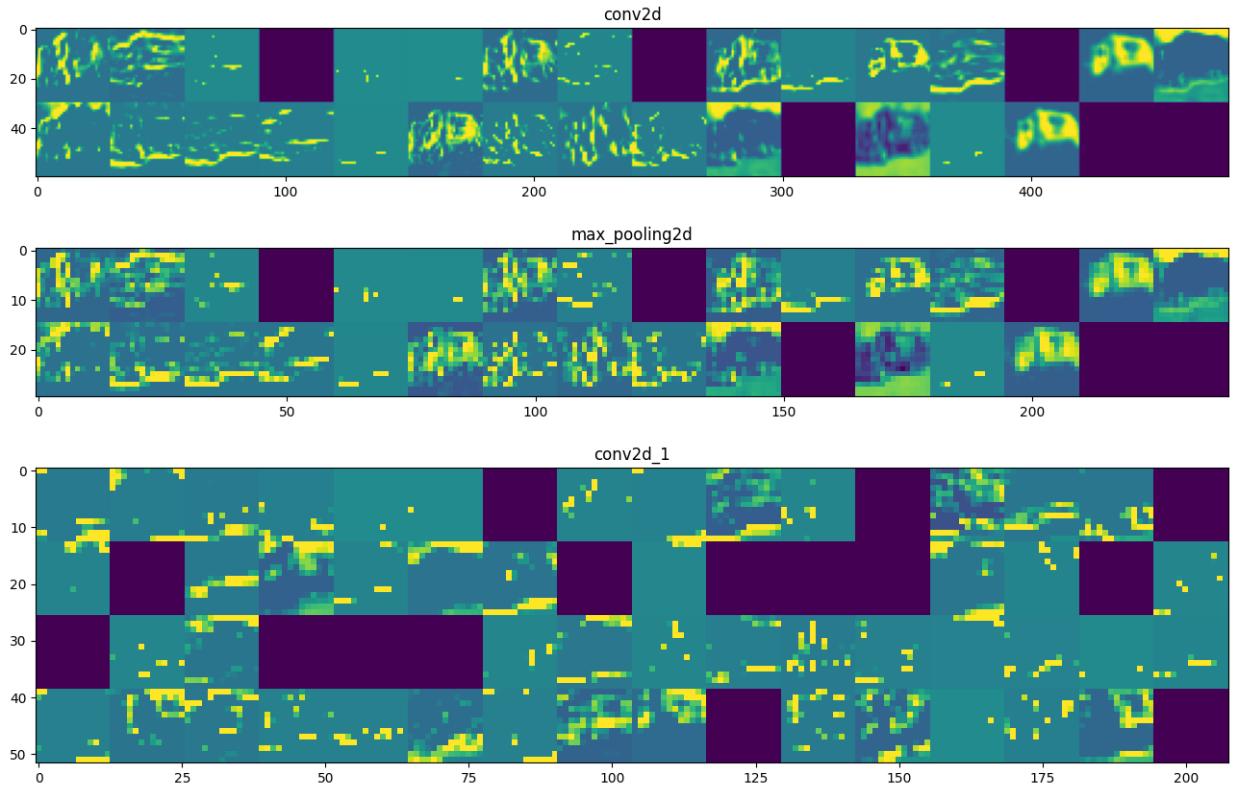
plt.show();
```



```
1/1 [=====] - 0s 71ms/step
```

```
<ipython-input-220-8e6847644768>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



7) Model 6 - CNN with 3 Max Pooling / Hidden Layers and Early Stopping and Batch Normalization Regularization

7.1) Build The Model

We use a Sequential class defined in Keras to create our model.

```
In [ ]: k.clear_session()
model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu),
    MaxPool2D((2, 2),strides=2),
    Flatten(),
    Dense(units=256,activation=tf.nn.softmax),
    BatchNormalization(),
    Dense(units=10, activation=tf.nn.softmax)
])
```

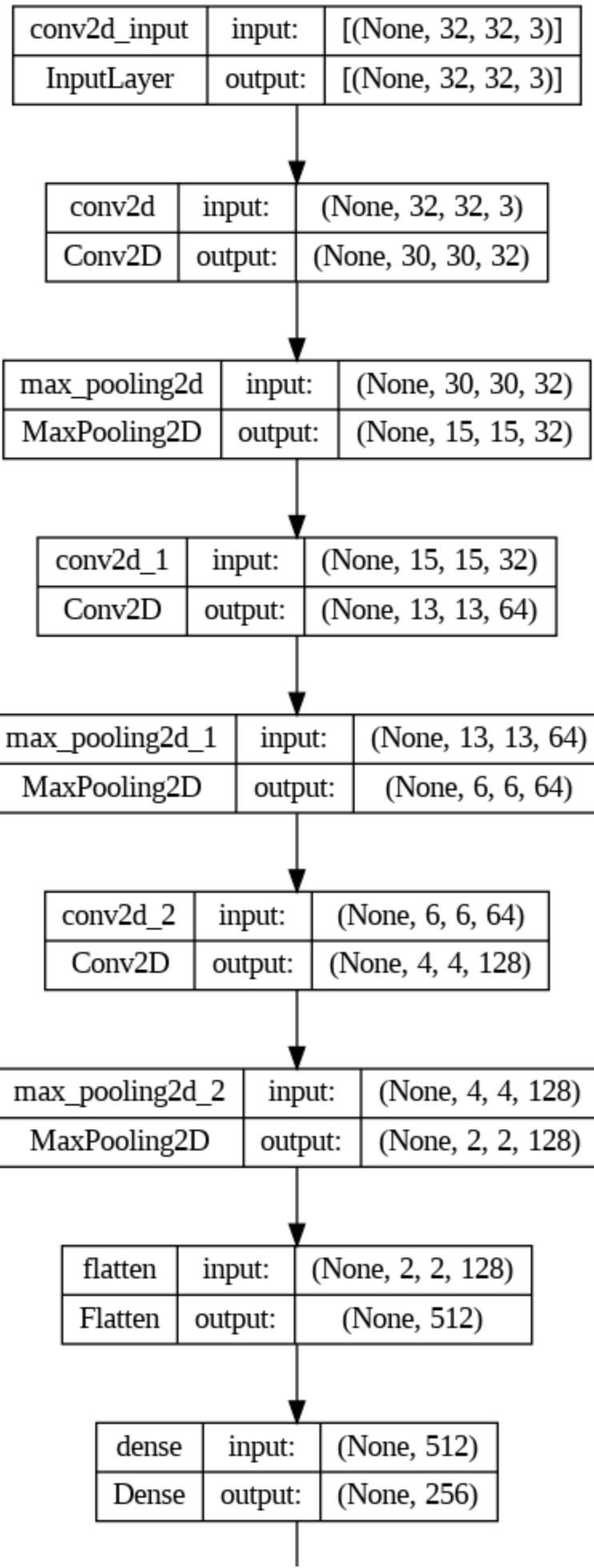
```
In [ ]: model.summary()
```

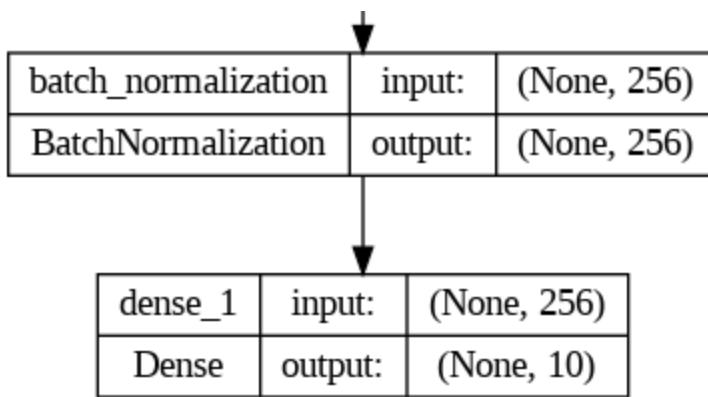
Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 256)	131328
batch_normalization (Batch Normalization)	(None, 256)	1024
dense_1 (Dense)	(None, 10)	2570
<hr/>		
Total params: 228170 (891.29 KB)		
Trainable params: 227658 (889.29 KB)		
Non-trainable params: 512 (2.00 KB)		

```
In [ ]: tf.keras.utils.plot_model(model, "CIFAR10.png", show_shapes=True)
```

Out[]:





Let's now compile and train the model.

tf.keras.losses.SparseCategoricalCrossentropy

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Module: tf.keras.callbacks

tf.keras.callbacks.EarlyStopping

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

tf.keras.callbacks.ModelCheckpoint

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

```
In [ ]: model.compile(optimizer='adam',
                      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
                      metrics=['accuracy'])
```

```
In [ ]: history = model.fit(x_train_norm
                           ,y_train_split
                           ,epochs=30
                           ,batch_size=500
                           ,validation_data=(x_valid_norm, y_valid_split)
                           ,callbacks=[tf.keras.callbacks.ModelCheckpoint("Model_6", save_best_only=True, s
                           ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=10)
                           ])
```

```
Epoch 1/30
90/90 [=====] - 59s 618ms/step - loss: 1.8476 - accuracy: 0.
3258 - val_loss: 2.2293 - val_accuracy: 0.1054
Epoch 2/30
90/90 [=====] - 37s 409ms/step - loss: 1.4585 - accuracy: 0.
4678 - val_loss: 2.1547 - val_accuracy: 0.1300
Epoch 3/30
90/90 [=====] - 38s 422ms/step - loss: 1.2940 - accuracy: 0.
5362 - val_loss: 2.0113 - val_accuracy: 0.2698
Epoch 4/30
90/90 [=====] - 39s 418ms/step - loss: 1.1729 - accuracy: 0.
5839 - val_loss: 1.7790 - val_accuracy: 0.4576
Epoch 5/30
90/90 [=====] - 38s 423ms/step - loss: 1.0814 - accuracy: 0.
6190 - val_loss: 1.4917 - val_accuracy: 0.5710
Epoch 6/30
90/90 [=====] - 38s 420ms/step - loss: 1.0176 - accuracy: 0.
6426 - val_loss: 1.2489 - val_accuracy: 0.5934
Epoch 7/30
90/90 [=====] - 37s 414ms/step - loss: 0.9588 - accuracy: 0.
6626 - val_loss: 1.1470 - val_accuracy: 0.6100
Epoch 8/30
90/90 [=====] - 36s 405ms/step - loss: 0.9096 - accuracy: 0.
6810 - val_loss: 1.1552 - val_accuracy: 0.5992
Epoch 9/30
90/90 [=====] - 37s 409ms/step - loss: 0.8664 - accuracy: 0.
6964 - val_loss: 1.1268 - val_accuracy: 0.6130
Epoch 10/30
90/90 [=====] - 36s 399ms/step - loss: 0.8220 - accuracy: 0.
7118 - val_loss: 1.0106 - val_accuracy: 0.6574
Epoch 11/30
90/90 [=====] - 37s 410ms/step - loss: 0.7738 - accuracy: 0.
7284 - val_loss: 0.9863 - val_accuracy: 0.6578
Epoch 12/30
90/90 [=====] - 37s 415ms/step - loss: 0.7371 - accuracy: 0.
7430 - val_loss: 0.9883 - val_accuracy: 0.6664
Epoch 13/30
90/90 [=====] - 37s 413ms/step - loss: 0.7108 - accuracy: 0.
7490 - val_loss: 0.9497 - val_accuracy: 0.6808
Epoch 14/30
90/90 [=====] - 37s 415ms/step - loss: 0.6806 - accuracy: 0.
7600 - val_loss: 0.9358 - val_accuracy: 0.6872
Epoch 15/30
90/90 [=====] - 37s 409ms/step - loss: 0.6458 - accuracy: 0.
7742 - val_loss: 0.9585 - val_accuracy: 0.6826
Epoch 16/30
90/90 [=====] - 35s 391ms/step - loss: 0.6096 - accuracy: 0.
7864 - val_loss: 0.9647 - val_accuracy: 0.6878
Epoch 17/30
90/90 [=====] - 37s 410ms/step - loss: 0.5819 - accuracy: 0.
7954 - val_loss: 0.9316 - val_accuracy: 0.6970
Epoch 18/30
90/90 [=====] - 36s 403ms/step - loss: 0.5549 - accuracy: 0.
8086 - val_loss: 0.9815 - val_accuracy: 0.6858
Epoch 19/30
90/90 [=====] - 37s 411ms/step - loss: 0.5301 - accuracy: 0.
8147 - val_loss: 0.9191 - val_accuracy: 0.7080
Epoch 20/30
90/90 [=====] - 38s 427ms/step - loss: 0.5056 - accuracy: 0.
8233 - val_loss: 1.0329 - val_accuracy: 0.6804
```

```
Epoch 21/30
90/90 [=====] - 36s 398ms/step - loss: 0.4799 - accuracy: 0.
8319 - val_loss: 0.9800 - val_accuracy: 0.6972
Epoch 22/30
90/90 [=====] - 36s 403ms/step - loss: 0.4595 - accuracy: 0.
8377 - val_loss: 0.9690 - val_accuracy: 0.7062
```

7.2) Evaluate Model Performance

```
In [ ]: model = tf.keras.models.load_model("Model_6")
print(f"Training accuracy: {model.evaluate(x_train_norm, y_train_split)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_norm, y_valid_split)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_norm, y_test)[1]:.3f}")

1407/1407 [=====] - 13s 9ms/step - loss: 0.5043 - accuracy:
0.8253
Training accuracy: 0.825
157/157 [=====] - 1s 9ms/step - loss: 0.9191 - accuracy: 0.7
080
Validation accuracy: 0.708
313/313 [=====] - 3s 9ms/step - loss: 0.9378 - accuracy: 0.6
940
Test accuracy: 0.694
```

```
In [ ]: preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)

313/313 [=====] - 3s 9ms/step
shape of preds: (10000, 10)
```

```
In [ ]: history_dict = history.history
history_dict.keys()

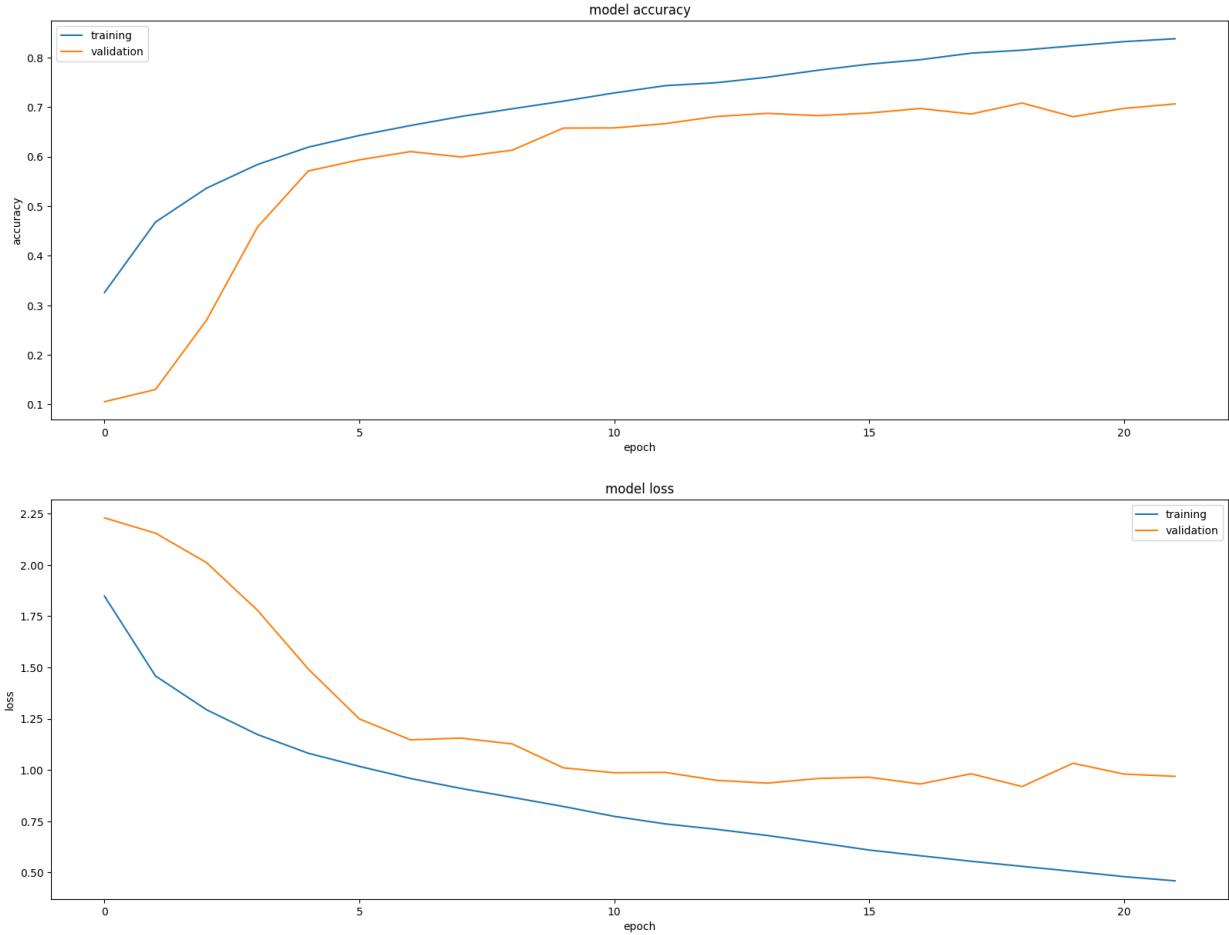
Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)
```

```
Out[ ]:
```

	loss	accuracy	val_loss	val_accuracy
17	0.555	0.809	0.981	0.686
18	0.530	0.815	0.919	0.708
19	0.506	0.823	1.033	0.680
20	0.480	0.832	0.980	0.697
21	0.459	0.838	0.969	0.706

```
In [ ]: plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
:17: MatplotlibDeprecationWarning: Auto-removal of over
lapping axes is deprecated since 3.6 and will be removed two minor releases later; ex
plicitly call ax.remove() as needed.
ax = plt.subplot(subplot)
```



Let's examine the precision, recall, F1 score, and confusion matrix.

```
In [ ]: pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)
```

```
313/313 [=====] - 3s 9ms/step
```

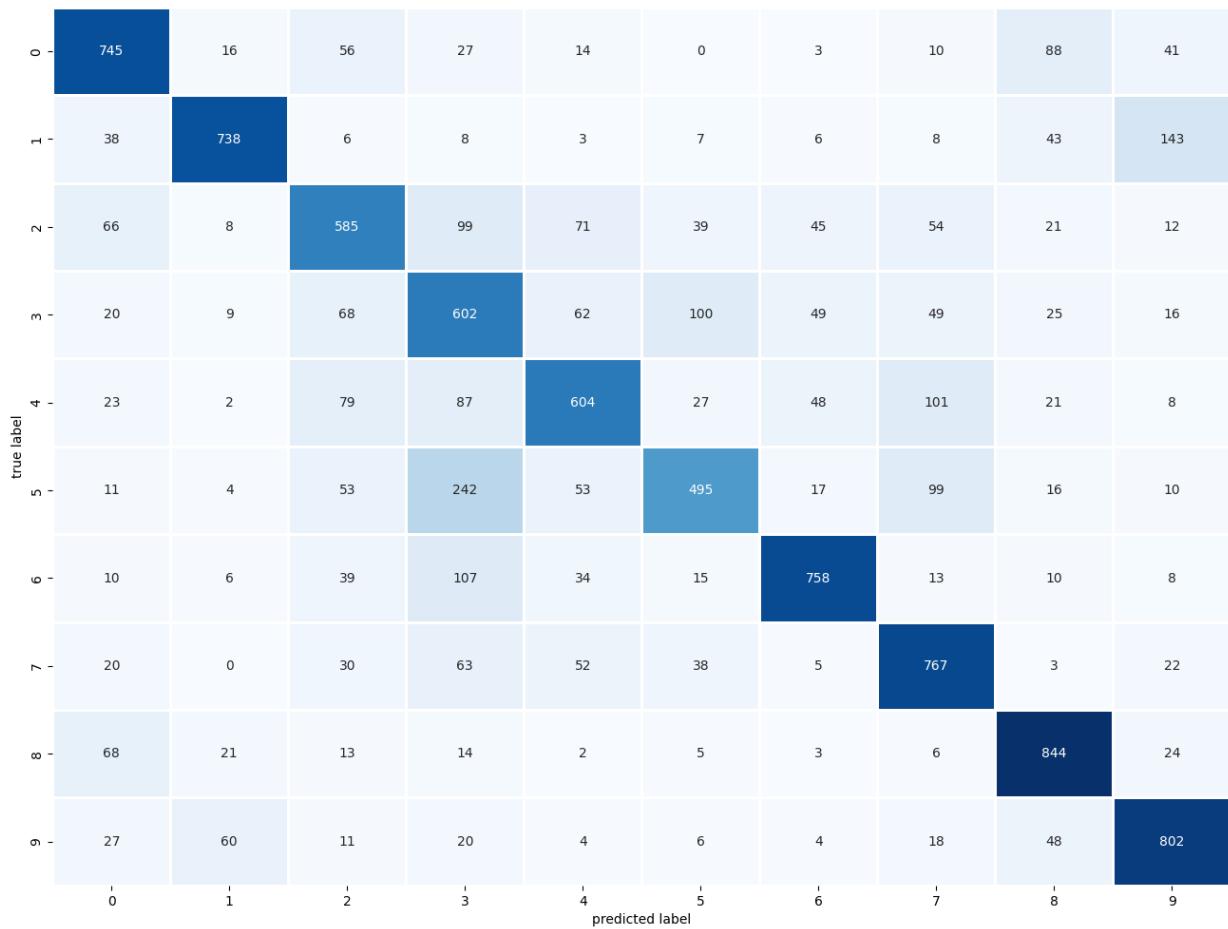
```
In [ ]: print_validation_report(y_test, pred1)
```

Classification Report				
	precision	recall	f1-score	support
0	0.72	0.74	0.73	1000
1	0.85	0.74	0.79	1000
2	0.62	0.58	0.60	1000
3	0.47	0.60	0.53	1000
4	0.67	0.60	0.64	1000
5	0.68	0.49	0.57	1000
6	0.81	0.76	0.78	1000
7	0.68	0.77	0.72	1000
8	0.75	0.84	0.80	1000
9	0.74	0.80	0.77	1000
accuracy			0.69	10000
macro avg	0.70	0.69	0.69	10000
weighted avg	0.70	0.69	0.69	10000

Accuracy Score: 0.694

Root Mean Square Error: 2.335636958090876

In []: `plot_confusion_matrix(y_test,pred1)`



Load HDF5 Model Format

tf.keras.models.load_model

https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model

In []: `model = tf.keras.models.load_model('Model_6')
preds = model.predict(x_test_norm)
preds.shape`

313/313 [=====] - 3s 9ms/step

Out[]: `(10000, 10)`

Let's examine the predictions for the testing dataset.

In []: `cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)

df = pd.DataFrame(preds[0:20], columns = ['airplane'
,'automobile'
, 'bird'
, 'cat'
, 'deer'
, 'dog'
, 'frog'
, 'horse'`

```

        , 'ship'
        , 'truck'])
df.style.format("{:.2%}").background_gradient(cmap=cm)

```

Out[]:

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0	0.05%	0.14%	0.28%	96.00%	0.07%	2.35%	0.66%	0.08%	0.32%	0.05%
1	5.84%	7.59%	0.01%	0.01%	0.00%	0.00%	0.00%	0.00%	86.41%	0.13%
2	1.18%	1.59%	0.04%	0.10%	0.01%	0.02%	0.03%	0.02%	95.33%	1.67%
3	96.19%		0.01%	0.04%	0.00%	0.01%	0.00%	0.00%	3.72%	0.02%
4	0.00%	0.00%	0.58%	3.59%	24.90%	0.15%	70.74%	0.01%	0.02%	0.00%
5	1.51%	1.12%	2.16%	32.56%	3.09%	3.49%	50.80%	1.35%	1.90%	2.03%
6	0.18%	90.57%	0.09%	6.76%	0.00%	1.36%	0.00%	0.01%	0.02%	1.00%
7	1.14%	0.01%	41.87%	2.10%	1.09%	0.18%	53.41%	0.11%	0.05%	0.03%
8	0.00%	0.00%	0.13%	93.20%	1.40%	4.05%	0.18%	1.03%	0.00%	0.01%
9	0.09%	98.98%	0.02%	0.00%	0.00%	0.00%	0.01%	0.00%	0.09%	0.82%
10	20.91%	0.02%	5.08%	24.43%	31.50%	8.31%	0.17%	3.13%	6.22%	0.23%
11	0.01%	0.04%	0.00%	0.01%	0.00%	0.00%	0.02%	0.00%	0.31%	99.60%
12	0.39%	0.21%	9.89%	28.89%	3.08%	34.64%	11.50%	10.88%	0.42%	0.10%
13	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%	0.00%	99.99%	0.00%	0.00%
14	0.00%	0.03%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	99.97%
15	0.75%	0.02%	0.41%	0.95%	0.02%	0.02%	31.08%	0.00%	66.73%	0.01%
16	0.19%	2.70%	5.16%	21.09%	0.11%	57.25%	0.21%	8.50%	3.31%	1.47%
17	0.32%	0.21%	1.37%	12.54%	2.42%	6.44%	1.29%	70.44%	0.33%	4.65%
18	9.36%	0.27%	0.03%	0.02%	0.02%	0.01%	0.02%	0.03%	31.32%	58.90%
19	0.00%	0.37%	0.00%	0.00%	0.28%	0.00%	99.33%	0.00%	0.00%	0.01%

In []:

```

(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[98]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
               , 'automobile'
               , 'bird'
               , 'cat'
               , 'deer'
               , 'dog'
               , 'frog'
               , 'horse'
               , 'ship'
               , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')

```

```
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

# Display the grid
```

```
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                    scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

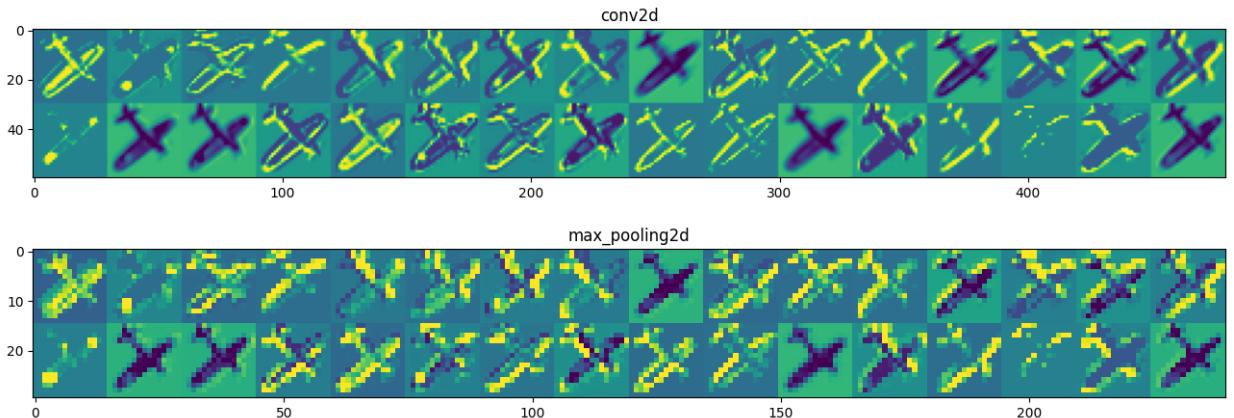
plt.show();
```

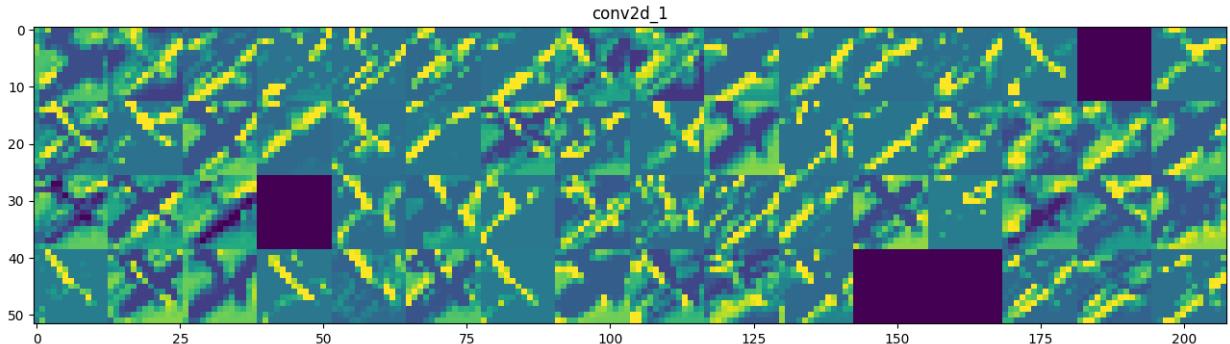


1/1 [=====] - 0s 107ms/step

<ipython-input-240-9ab60ebbedca>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[122]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)
```

```
images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

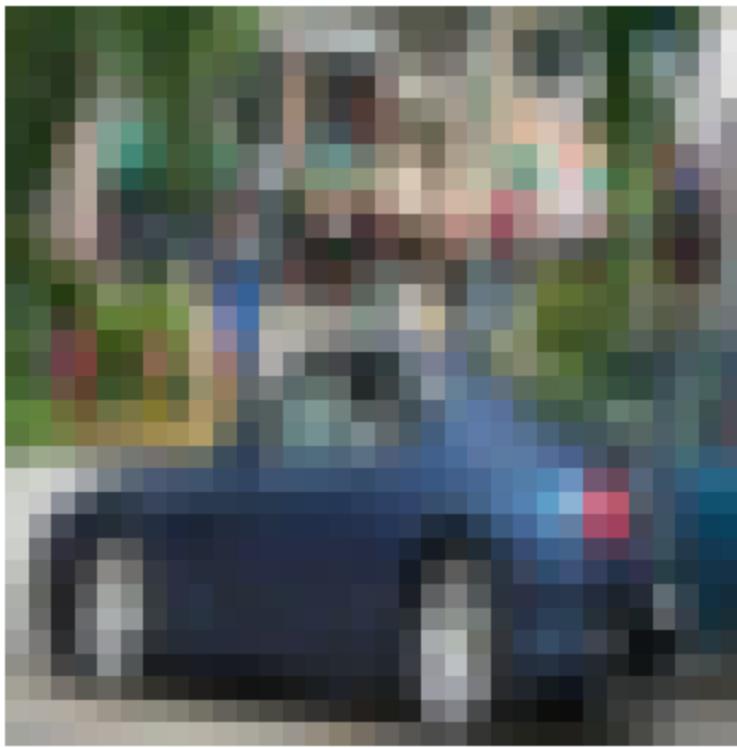
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

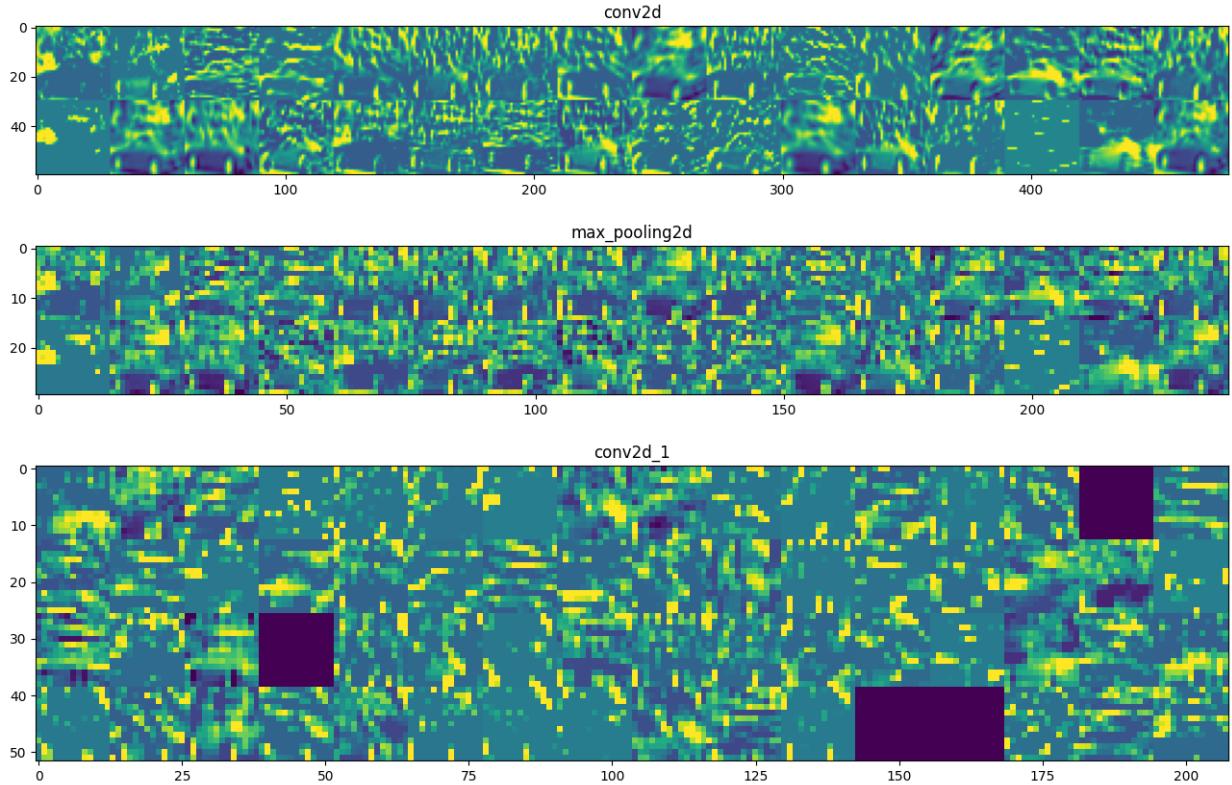
plt.show();
```



```
1/1 [=====] - 0s 94ms/step
```

```
<ipython-input-241-e0d043c5b9b7>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[75]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

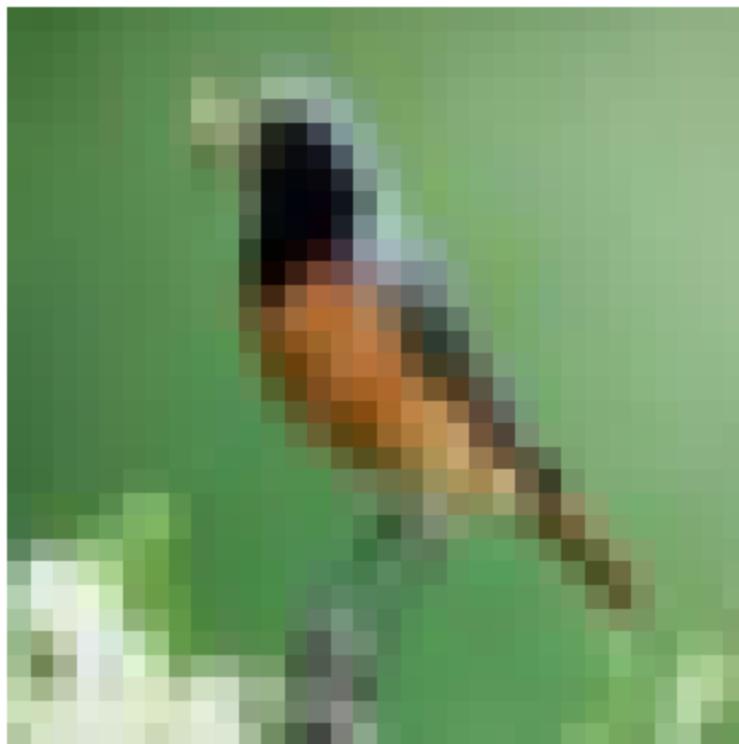
channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

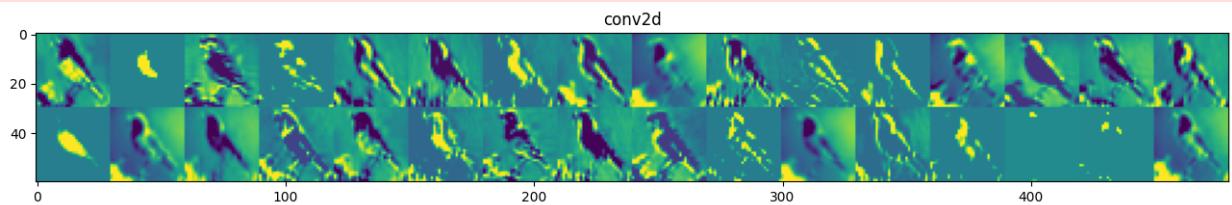
```

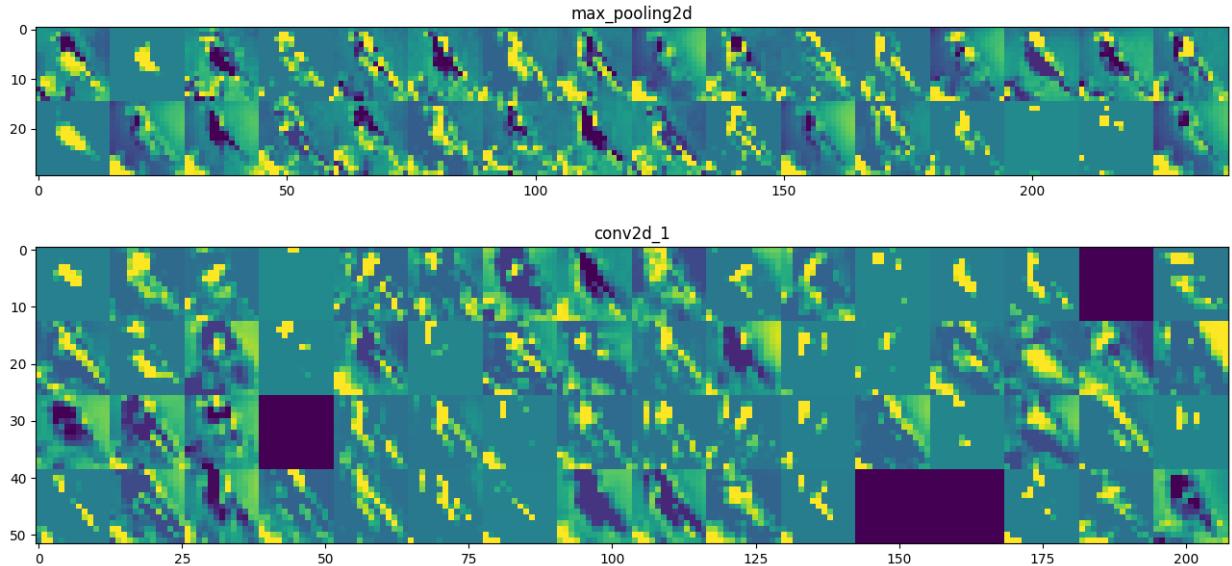


1/1 [=====] - 0s 94ms/step

<ipython-input-242-4848b71b261c>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[184]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

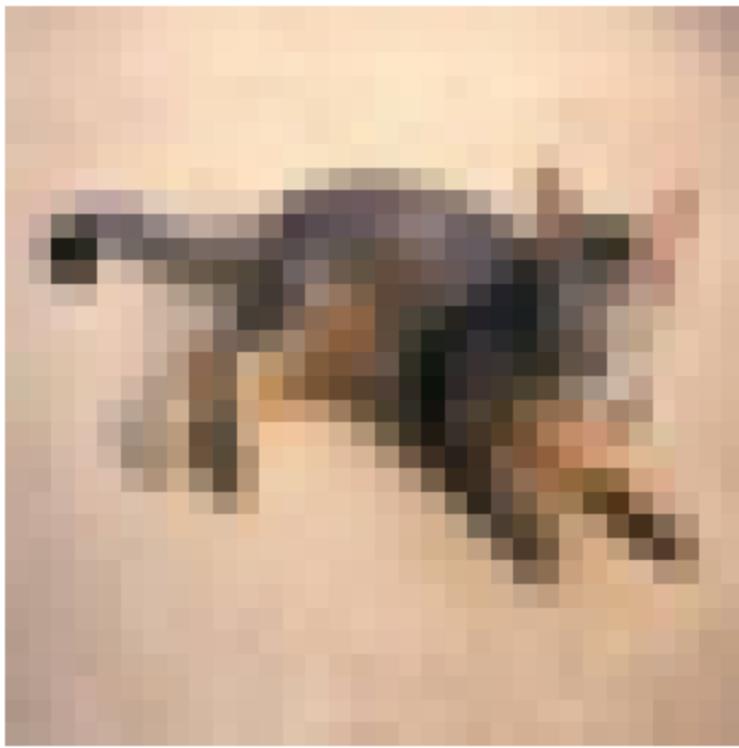
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

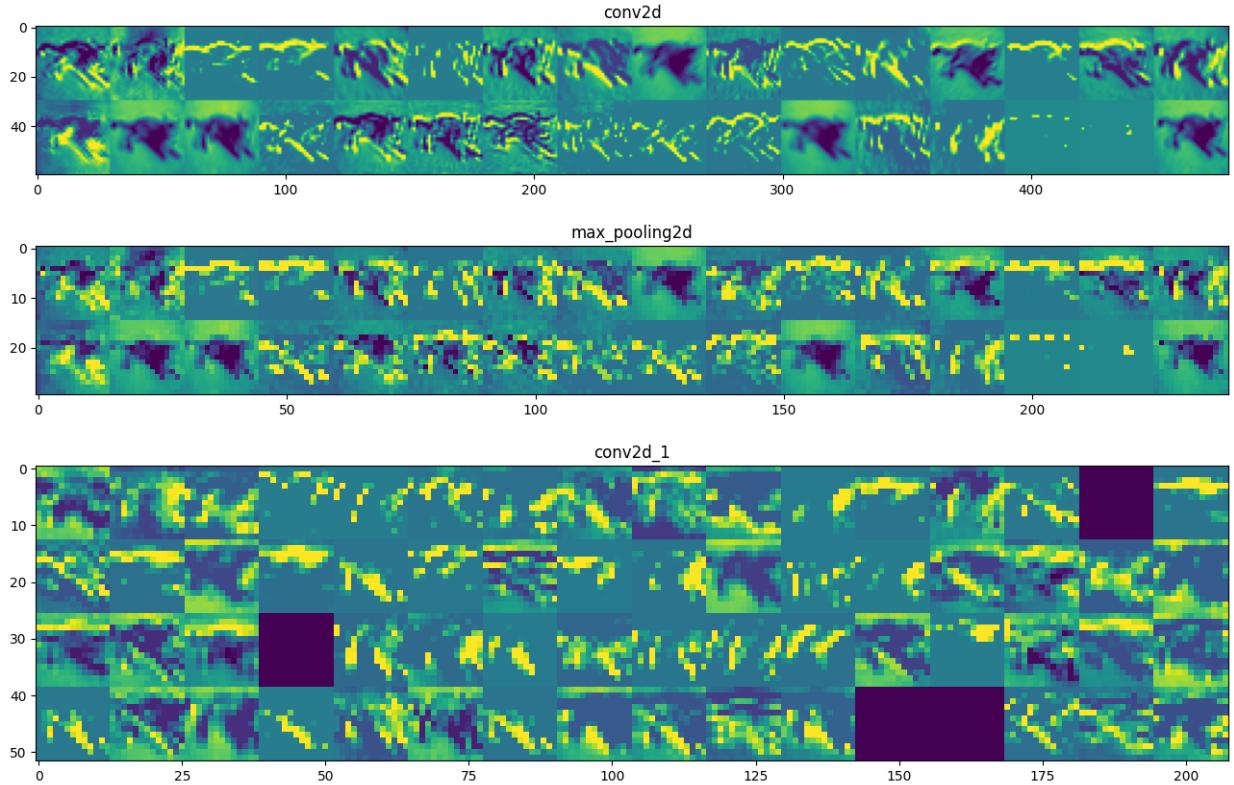
plt.show();
```



```
1/1 [=====] - 0s 91ms/step
```

```
<ipython-input-243-3bac8bdd9965>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[159]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

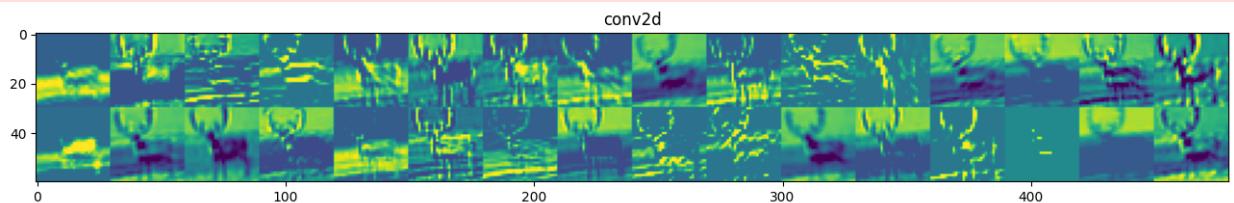
```

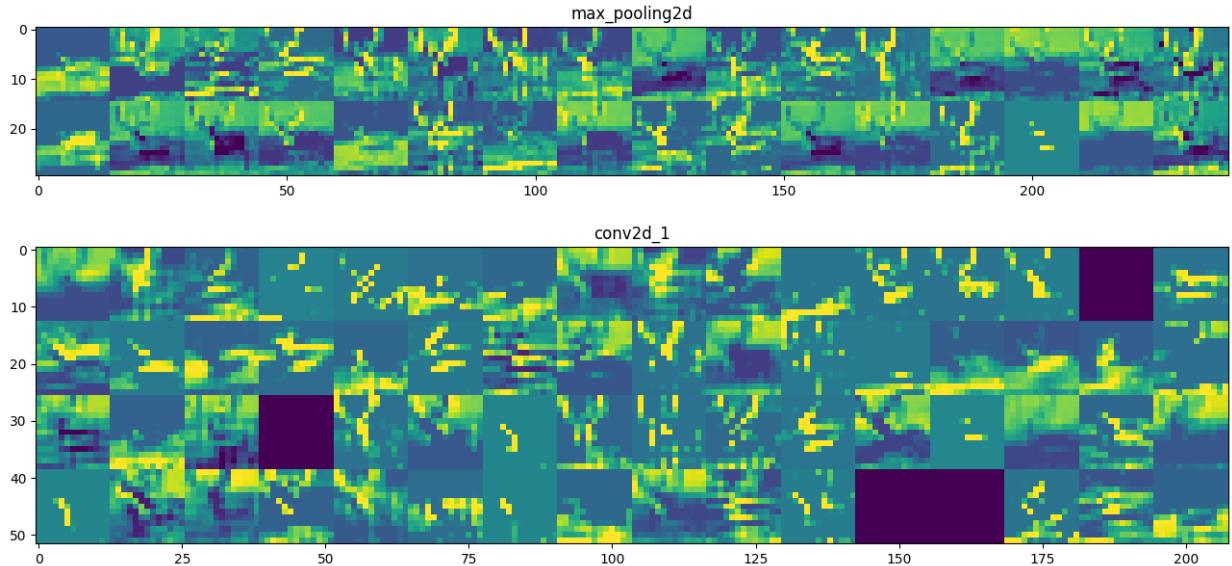


1/1 [=====] - 0s 85ms/step

<ipython-input-244-a52112e96c59>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[24]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

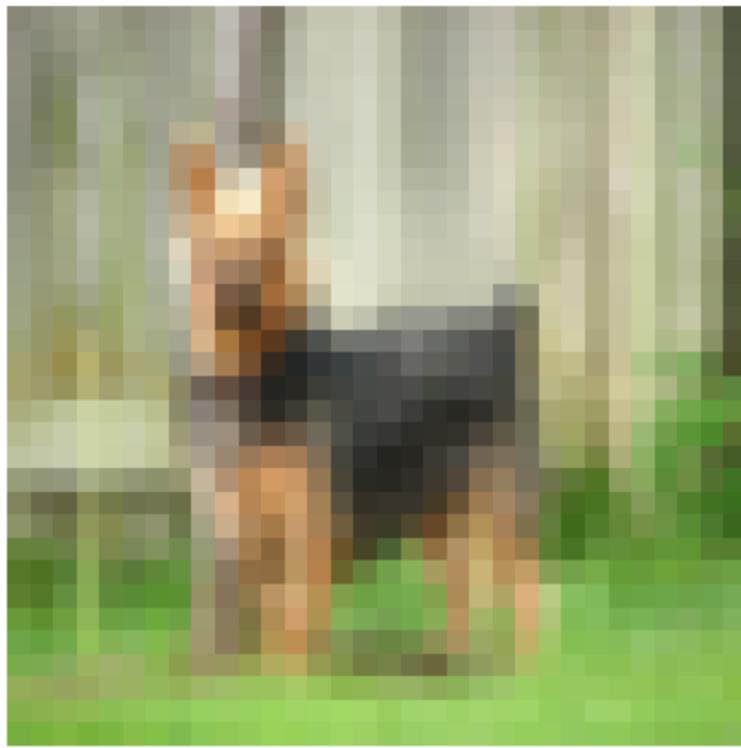
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

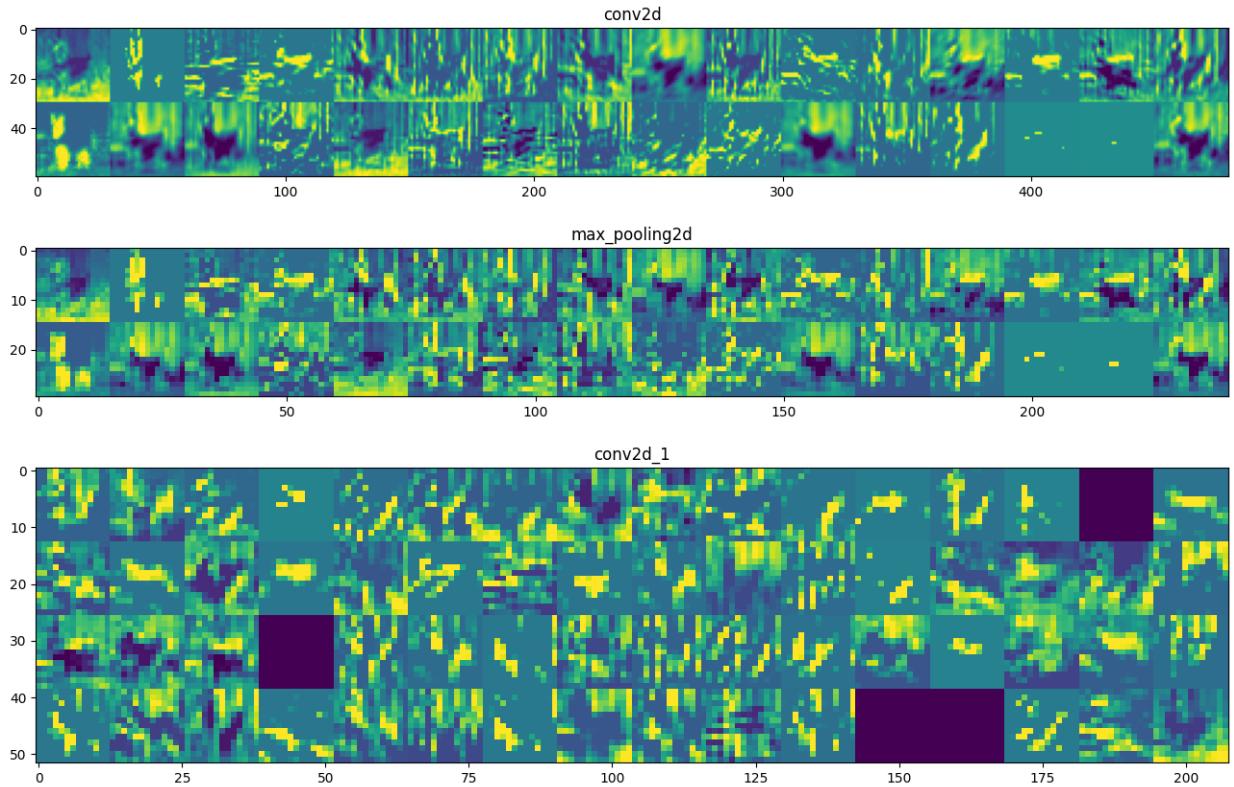
plt.show();
```



```
1/1 [=====] - 0s 81ms/step
```

```
<ipython-input-245-edee34e9d17b>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[152]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```

class_names = ['airplane'
,'automobile'
,'bird'
,'cat'
,'deer'
,'dog'
,'frog'
,'horse'
,'ship'
,'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):

```

```

channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
              row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

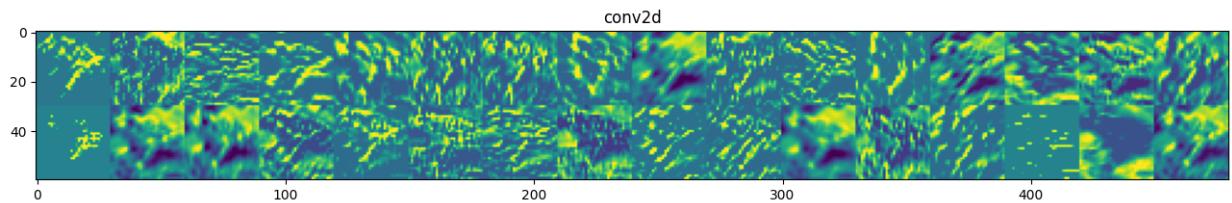
```

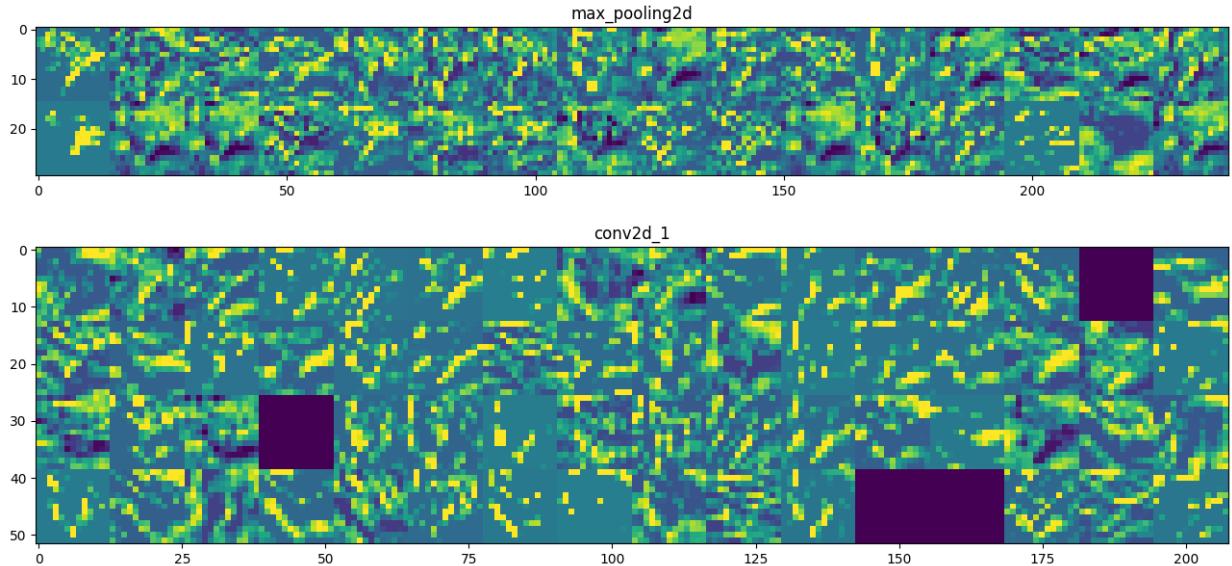


1/1 [=====] - 0s 80ms/step

<ipython-input-246-0ef3f9c61bfb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[2004]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

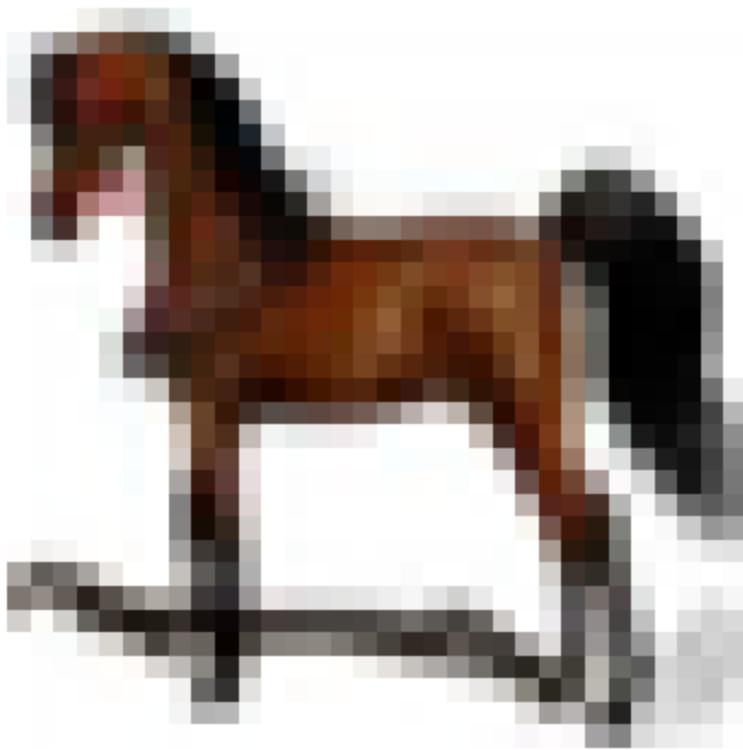
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

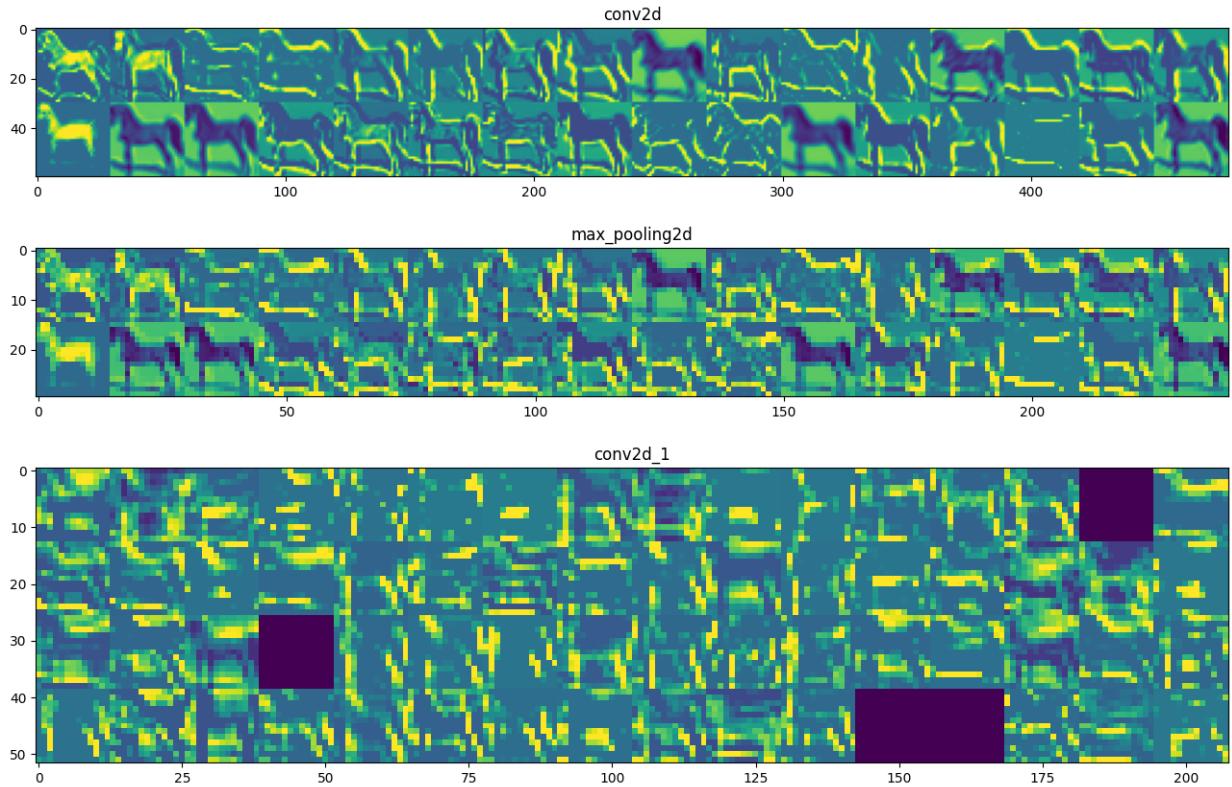
plt.show();
```



```
1/1 [=====] - 0s 83ms/step
```

```
<ipython-input-247-5c55fce06c66>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[185]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

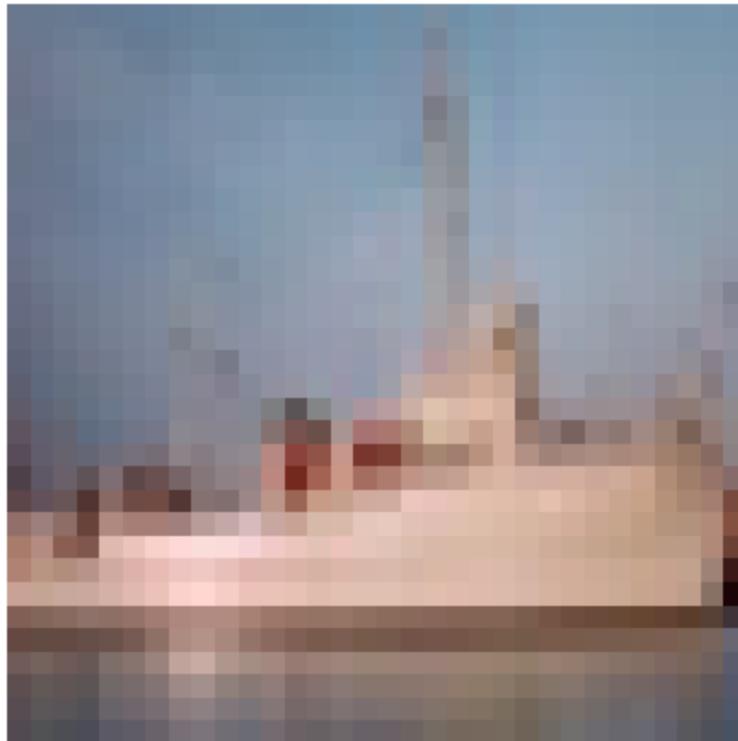
channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

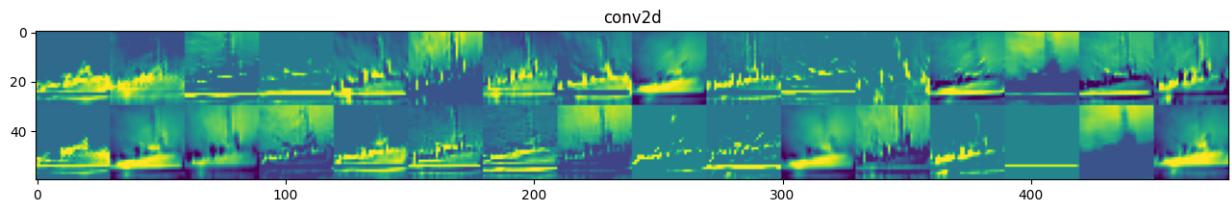
```

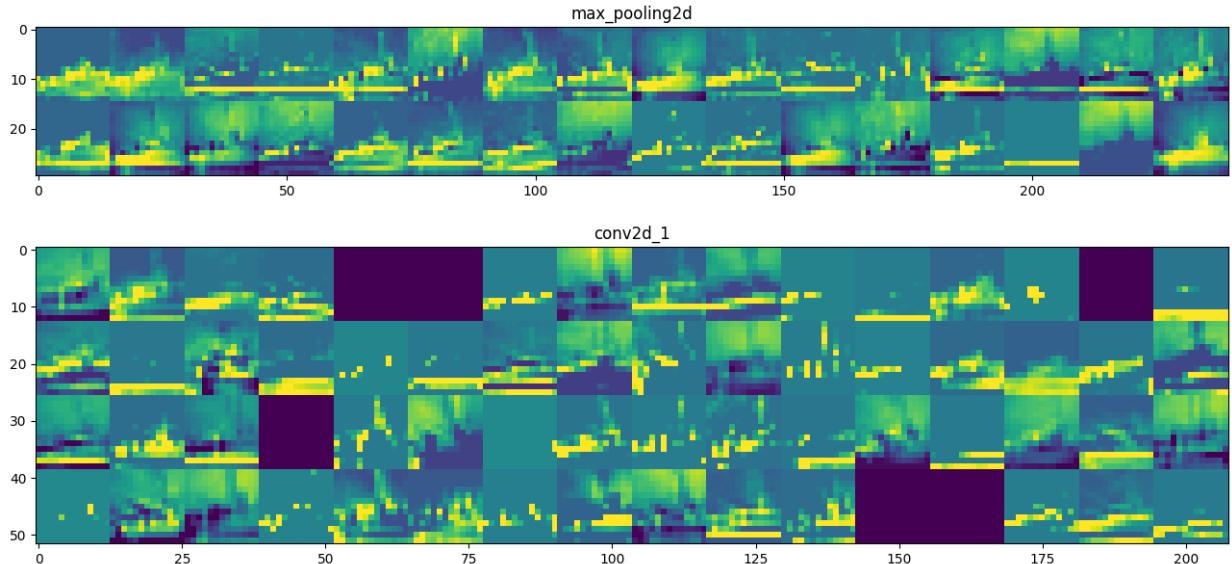


1/1 [=====] - 0s 85ms/step

<ipython-input-248-f532589aadeb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[133]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

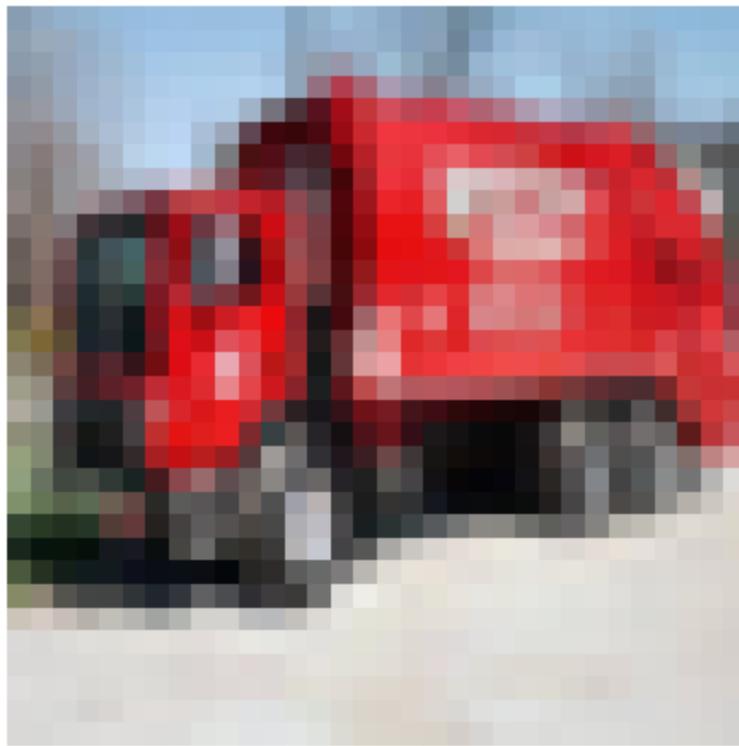
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

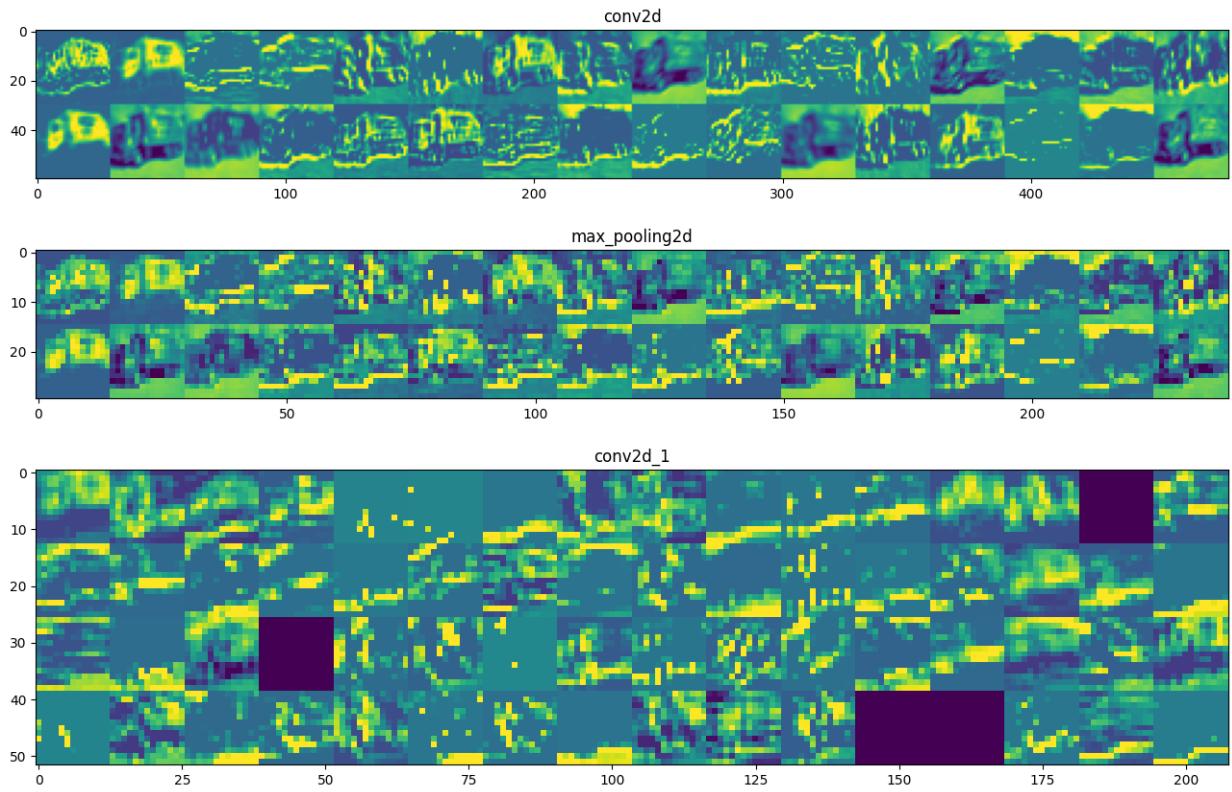
plt.show();
```



```
1/1 [=====] - 0s 115ms/step
```

```
<ipython-input-249-8e6847644768>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



In []:

8) Model 7 - CNN with 3 Max Pooling / Hidden Layers and Early Stopping, Batch Normalization, and L2 Regularization

8.1) Build The Model

We use a Sequential class defined in Keras to create our model.

```
In [ ]: k.clear_session()
model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_shape=(28, 28, 1)),
    MaxPool2D((2, 2), strides=2),
    Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_shape=(14, 14, 32)),
    MaxPool2D((2, 2), strides=2),
    Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu),
    MaxPool2D((2, 2), strides=2),
    Flatten(),
    Dense(units=256, activation=tf.nn.softmax, kernel_regularizer=tf.keras.regularizers.L2(0.01)),
    BatchNormalization(),
    Dense(units=10, activation=tf.nn.softmax)
])

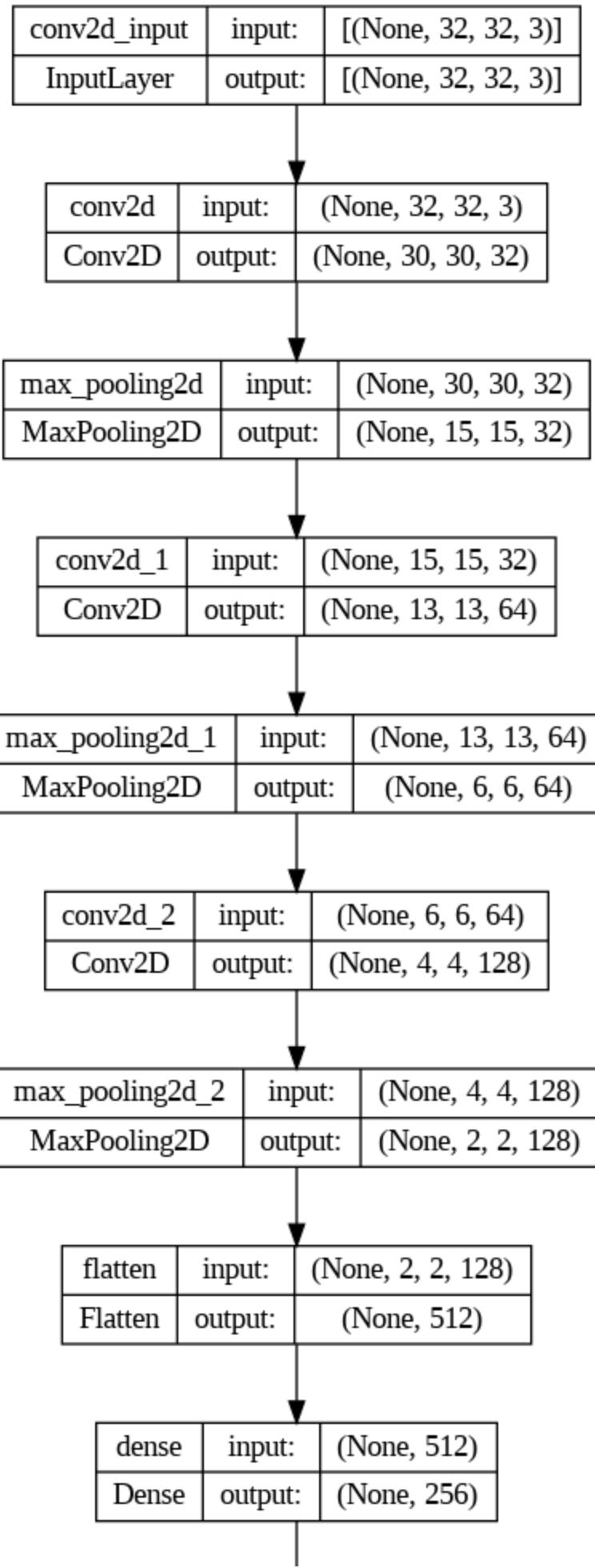
In [ ]: model.summary()
```

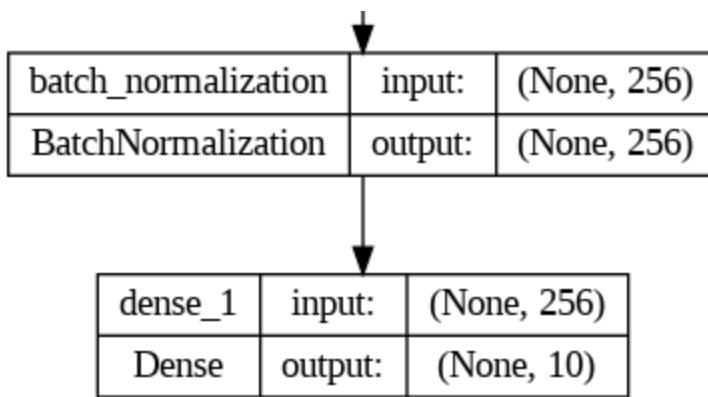
Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 256)	131328
batch_normalization (Batch Normalization)	(None, 256)	1024
dense_1 (Dense)	(None, 10)	2570
<hr/>		
Total params: 228170 (891.29 KB)		
Trainable params: 227658 (889.29 KB)		
Non-trainable params: 512 (2.00 KB)		

In []: `tf.keras.utils.plot_model(model, "CIFAR10.png", show_shapes=True)`

Out[]:





Let's now compile and train the model.

tf.keras.losses.SparseCategoricalCrossentropy

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Module: tf.keras.callbacks

tf.keras.callbacks.EarlyStopping

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

tf.keras.callbacks.ModelCheckpoint

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

```
In [ ]: model.compile(optimizer='adam',
                      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
                      metrics=['accuracy'])
```

```
In [ ]: history = model.fit(x_train_norm
                           ,y_train_split
                           ,epochs=40
                           ,batch_size=500
                           ,validation_data=(x_valid_norm, y_valid_split)
                           ,callbacks=[
                           tf.keras.callbacks.ModelCheckpoint("Model_7", save_best_only=True,
                           ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=5)
                           ])
```

```
Epoch 1/40
90/90 [=====] - 54s 588ms/step - loss: 2.9286 - accuracy: 0.
2851 - val_loss: 2.3409 - val_accuracy: 0.1152
Epoch 2/40
90/90 [=====] - 42s 467ms/step - loss: 1.6107 - accuracy: 0.
4371 - val_loss: 2.1907 - val_accuracy: 0.1542
Epoch 3/40
90/90 [=====] - 37s 417ms/step - loss: 1.4183 - accuracy: 0.
5116 - val_loss: 2.0471 - val_accuracy: 0.3552
Epoch 4/40
90/90 [=====] - 37s 413ms/step - loss: 1.3110 - accuracy: 0.
5537 - val_loss: 1.8229 - val_accuracy: 0.4620
Epoch 5/40
90/90 [=====] - 37s 417ms/step - loss: 1.2299 - accuracy: 0.
5823 - val_loss: 1.6528 - val_accuracy: 0.5008
Epoch 6/40
90/90 [=====] - 37s 414ms/step - loss: 1.1702 - accuracy: 0.
6072 - val_loss: 1.3407 - val_accuracy: 0.5786
Epoch 7/40
90/90 [=====] - 38s 422ms/step - loss: 1.1119 - accuracy: 0.
6300 - val_loss: 1.2369 - val_accuracy: 0.5924
Epoch 8/40
90/90 [=====] - 37s 410ms/step - loss: 1.0572 - accuracy: 0.
6479 - val_loss: 1.1701 - val_accuracy: 0.6086
Epoch 9/40
90/90 [=====] - 36s 394ms/step - loss: 1.0213 - accuracy: 0.
6622 - val_loss: 1.1809 - val_accuracy: 0.6118
Epoch 10/40
90/90 [=====] - 37s 413ms/step - loss: 0.9887 - accuracy: 0.
6740 - val_loss: 1.1078 - val_accuracy: 0.6376
Epoch 11/40
90/90 [=====] - 36s 402ms/step - loss: 0.9579 - accuracy: 0.
6862 - val_loss: 1.1665 - val_accuracy: 0.6182
Epoch 12/40
90/90 [=====] - 36s 405ms/step - loss: 0.9259 - accuracy: 0.
6985 - val_loss: 1.1079 - val_accuracy: 0.6438
Epoch 13/40
90/90 [=====] - 37s 409ms/step - loss: 0.8932 - accuracy: 0.
7120 - val_loss: 1.1634 - val_accuracy: 0.6332
Epoch 14/40
90/90 [=====] - 35s 390ms/step - loss: 0.8777 - accuracy: 0.
7164 - val_loss: 1.2754 - val_accuracy: 0.6160
Epoch 15/40
90/90 [=====] - 37s 415ms/step - loss: 0.8573 - accuracy: 0.
7241 - val_loss: 1.0745 - val_accuracy: 0.6544
Epoch 16/40
90/90 [=====] - 40s 442ms/step - loss: 0.8252 - accuracy: 0.
7352 - val_loss: 1.1284 - val_accuracy: 0.6532
Epoch 17/40
90/90 [=====] - 37s 413ms/step - loss: 0.8139 - accuracy: 0.
7389 - val_loss: 1.1579 - val_accuracy: 0.6392
Epoch 18/40
90/90 [=====] - 41s 461ms/step - loss: 0.7902 - accuracy: 0.
7474 - val_loss: 1.0755 - val_accuracy: 0.6590
Epoch 19/40
90/90 [=====] - 36s 406ms/step - loss: 0.7692 - accuracy: 0.
7562 - val_loss: 1.0943 - val_accuracy: 0.6600
Epoch 20/40
90/90 [=====] - 37s 410ms/step - loss: 0.7519 - accuracy: 0.
7627 - val_loss: 1.2308 - val_accuracy: 0.6238
```

```
Epoch 21/40
90/90 [=====] - 37s 414ms/step - loss: 0.7364 - accuracy: 0.
7667 - val_loss: 1.1236 - val_accuracy: 0.6568
Epoch 22/40
90/90 [=====] - 36s 398ms/step - loss: 0.7239 - accuracy: 0.
7713 - val_loss: 1.1156 - val_accuracy: 0.6596
```

8.2) Evaluate Model Performance

```
In [ ]: model = tf.keras.models.load_model("Model_7")
print(f"Training accuracy: {model.evaluate(x_train_norm, y_train_split)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_norm, y_valid_split)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_norm, y_test)[1]:.3f}")

1407/1407 [=====] - 13s 9ms/step - loss: 0.8956 - accuracy:
0.7073
Training accuracy: 0.707
157/157 [=====] - 1s 8ms/step - loss: 1.0745 - accuracy: 0.6
544
Validation accuracy: 0.654
313/313 [=====] - 3s 9ms/step - loss: 1.0572 - accuracy: 0.6
599
Test accuracy: 0.660
```

```
In [ ]: preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)

313/313 [=====] - 3s 10ms/step
shape of preds: (10000, 10)
```

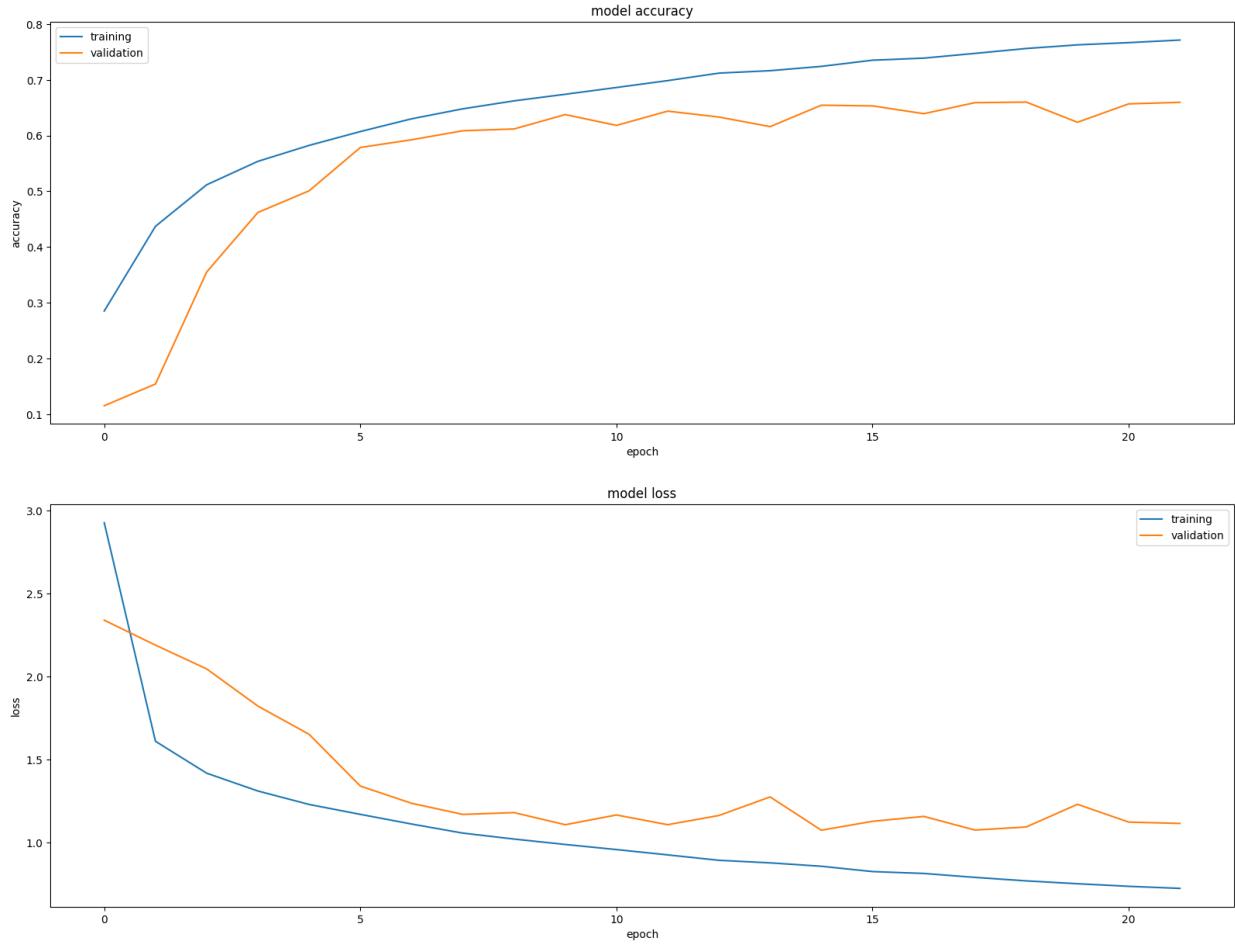
```
In [ ]: history_dict = history.history
history_dict.keys()

Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)
```

```
Out[ ]:   loss  accuracy  val_loss  val_accuracy
17  0.790      0.747     1.075      0.659
18  0.769      0.756     1.094      0.660
19  0.752      0.763     1.231      0.624
20  0.736      0.767     1.124      0.657
21  0.724      0.771     1.116      0.660
```

```
In [ ]: plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
<ipython-input-8-353fbbae40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of over
lapping axes is deprecated since 3.6 and will be removed two minor releases later; ex
plicitly call ax.remove() as needed.
    ax = plt.subplot(subplot)
```



Let's examine the precision, recall, F1 score, and confusion matrix.

```
In [ ]: pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

313/313 [=====] - 2s 8ms/step
```

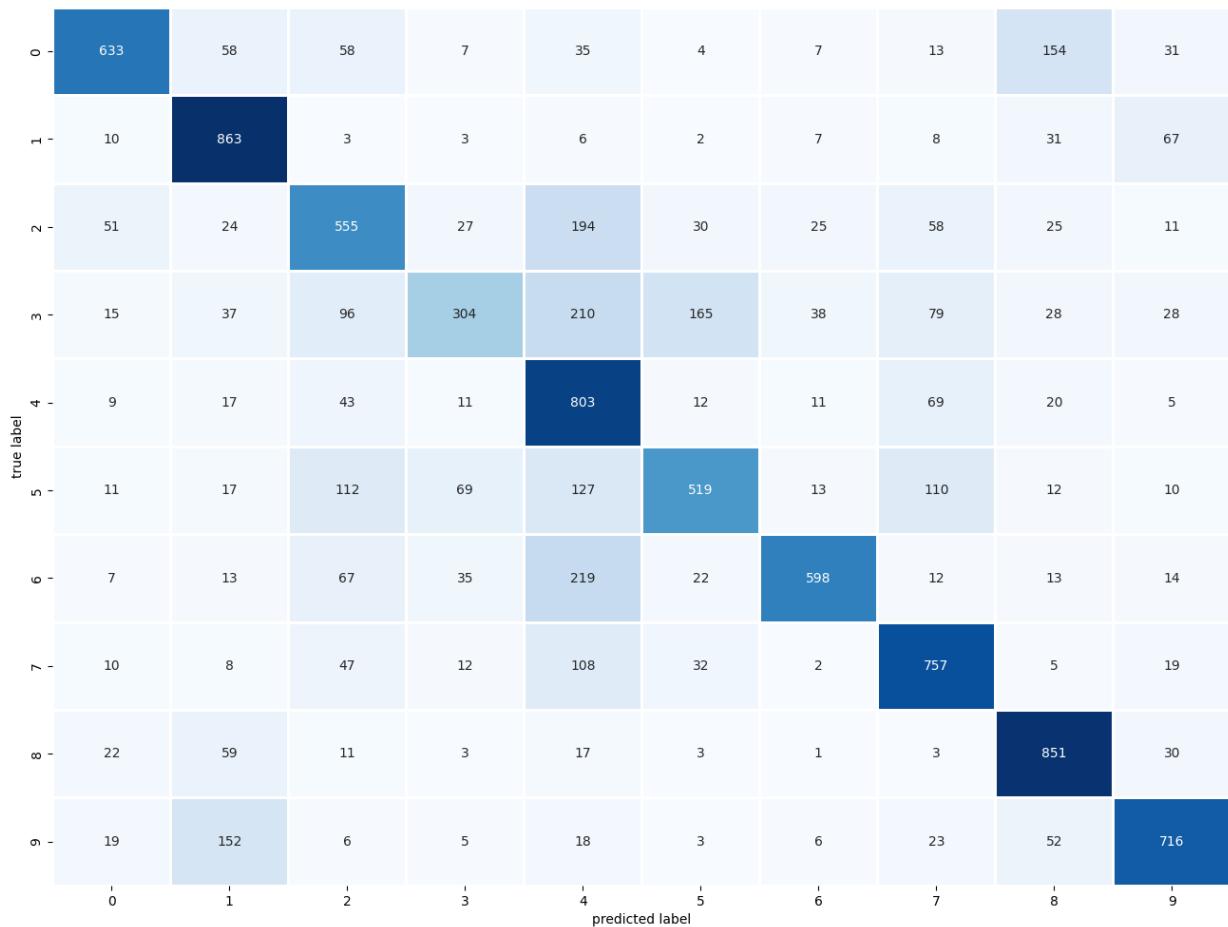
```
In [ ]: print_validation_report(y_test, pred1)
```

Classification Report				
	precision	recall	f1-score	support
0	0.80	0.63	0.71	1000
1	0.69	0.86	0.77	1000
2	0.56	0.56	0.56	1000
3	0.64	0.30	0.41	1000
4	0.46	0.80	0.59	1000
5	0.66	0.52	0.58	1000
6	0.84	0.60	0.70	1000
7	0.67	0.76	0.71	1000
8	0.71	0.85	0.78	1000
9	0.77	0.72	0.74	1000
accuracy			0.66	10000
macro avg	0.68	0.66	0.65	10000
weighted avg	0.68	0.66	0.65	10000

Accuracy Score: 0.6599

Root Mean Square Error: 2.4084642409635233

In []: `plot_confusion_matrix(y_test,pred1)`



Load HDF5 Model Format

`tf.keras.models.load_model`

https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model

In []: `model = tf.keras.models.load_model('Model_7')
preds = model.predict(x_test_norm)
preds.shape`

313/313 [=====] - 3s 8ms/step

Out[]: `(10000, 10)`

Let's examine the predictions for the testing dataset.

In []: `cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)`

```
df = pd.DataFrame(preds[0:20], columns = ['airplane'
                                         , 'automobile'
                                         , 'bird'
                                         , 'cat'
                                         , 'deer'
                                         , 'dog'
                                         , 'frog'
                                         , 'horse']
```

```

        , 'ship'
        , 'truck'])
df.style.format("{:.2%}").background_gradient(cmap=cm)

```

Out[]:

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0	0.03%	0.09%	0.36%	80.54%	3.58%	14.18%	0.87%	0.08%	0.16%	0.12%
1	24.08%	46.15%	0.01%	0.01%	0.00%	0.00%	0.00%	0.00%	29.01%	0.73%
2	5.39%	8.31%	0.09%	0.14%	0.06%	0.02%	0.06%	0.02%	78.35%	7.56%
3	52.02%	4.75%	0.61%	0.03%	0.86%	0.01%	0.07%	0.01%	41.22%	0.43%
4	0.01%	0.13%	8.61%	2.37%	43.82%	2.38%	42.62%	0.02%	0.03%	0.01%
5	0.10%	0.93%	1.80%	8.85%	5.89%	8.60%	69.68%	0.80%	0.91%	2.44%
6	0.50%	88.11%	0.08%	0.57%	0.00%	0.23%	0.01%	0.06%	0.00%	10.43%
7	1.86%	0.06%	28.20%	3.09%	47.61%	1.11%	16.38%	1.54%	0.10%	0.04%
8	0.12%	0.06%	8.52%	20.10%	17.79%	39.06%	0.69%	13.53%	0.06%	0.08%
9	0.12%	98.23%	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.17%	1.46%
10	27.25%	0.09%	21.38%	1.75%	31.75%	1.62%	0.09%	2.26%	13.59%	0.23%
11	0.01%	0.26%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.13%	99.60%
12	0.06%	0.09%	10.26%	12.01%	26.69%	46.02%	0.89%	3.83%	0.09%	0.06%
13	0.02%	0.00%	0.00%	0.00%	0.10%	0.07%	0.00%	99.79%	0.00%	0.02%
14	0.11%	1.92%	0.01%	0.00%	0.00%	0.00%	0.00%	0.02%	0.45%	97.48%
15	7.15%	0.69%	1.32%	2.34%	17.33%	0.26%	6.75%	0.03%	62.40%	1.74%
16	0.02%	0.38%	0.92%	12.08%	0.12%	85.24%	0.11%	0.54%	0.31%	0.28%
17	0.22%	0.87%	4.15%	7.44%	17.43%	9.94%	0.23%	56.43%	1.61%	1.68%
18	1.70%	1.25%	0.00%	0.01%	0.00%	0.00%	0.01%	0.00%	96.09%	0.94%
19	0.01%	0.21%	0.93%	3.68%	42.95%	4.89%	47.18%	0.13%	0.01%	0.01%

In []:

```

(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[98]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
               , 'automobile'
               , 'bird'
               , 'cat'
               , 'deer'
               , 'dog'
               , 'frog'
               , 'horse'
               , 'ship'
               , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')

```

```
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

# Display the grid
```

```
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                    scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

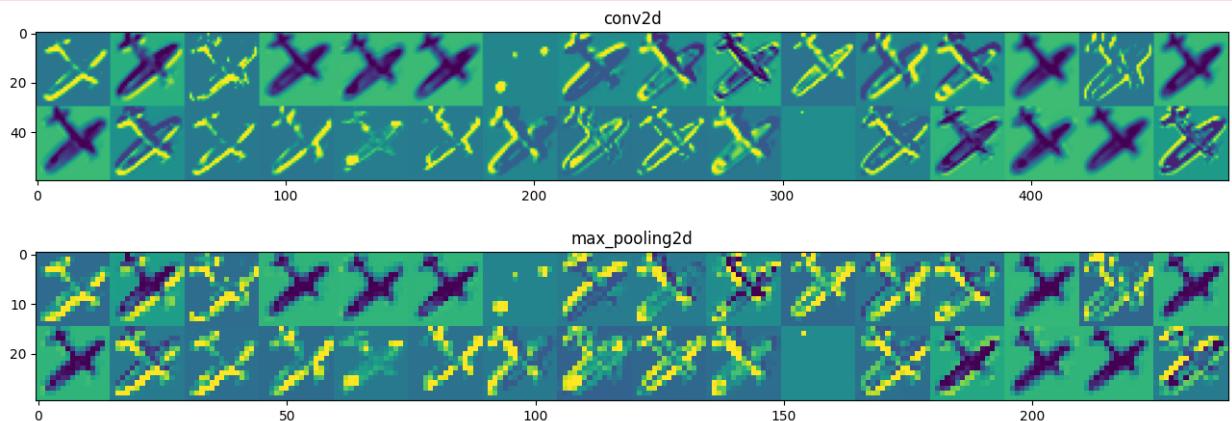
plt.show();
```

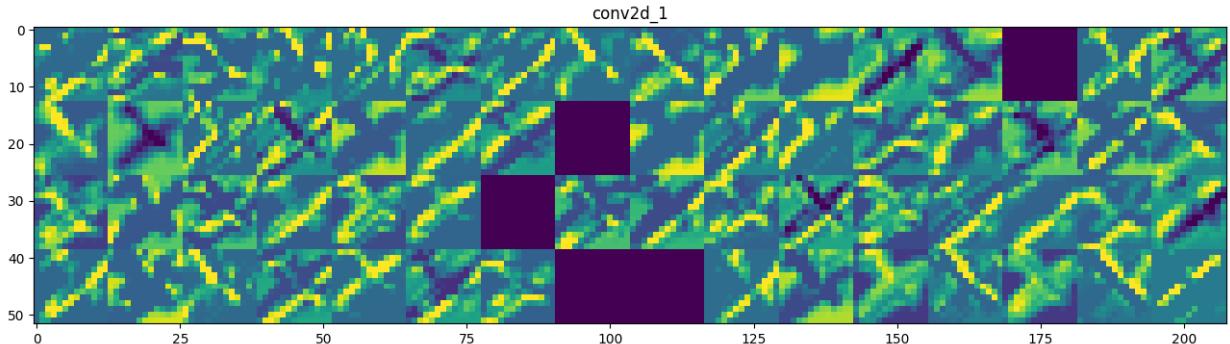


1/1 [=====] - 0s 68ms/step

<ipython-input-265-9ab60ebbedca>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[122]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)
```

```
images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

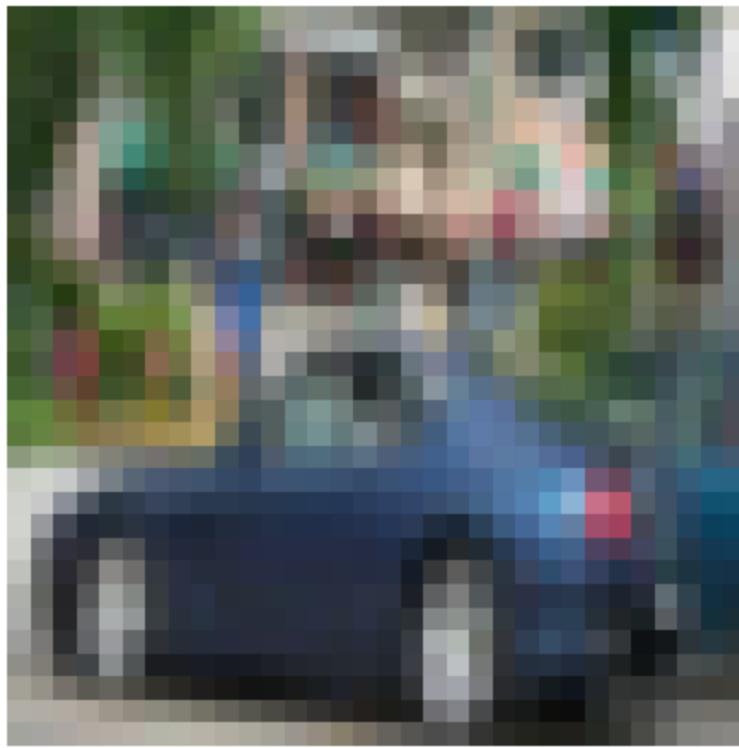
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

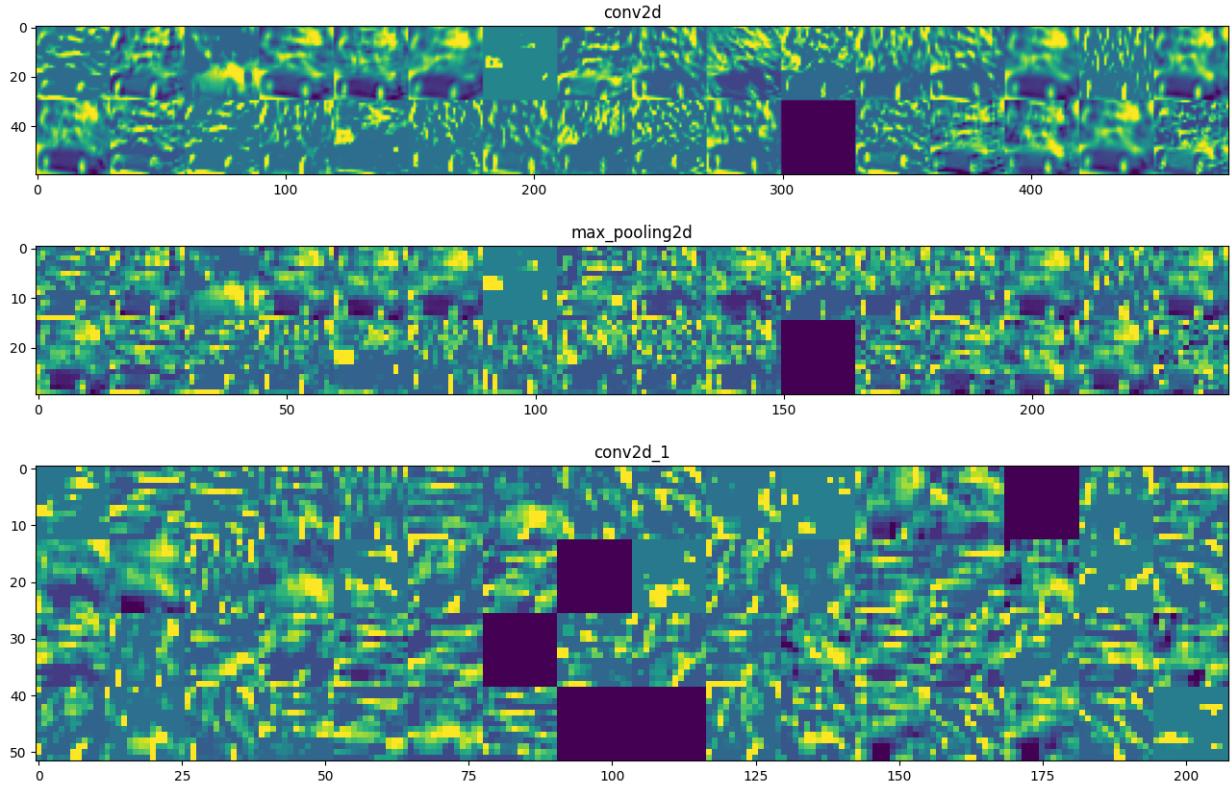
plt.show();
```



```
1/1 [=====] - 0s 83ms/step
```

```
<ipython-input-266-e0d043c5b9b7>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[75]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

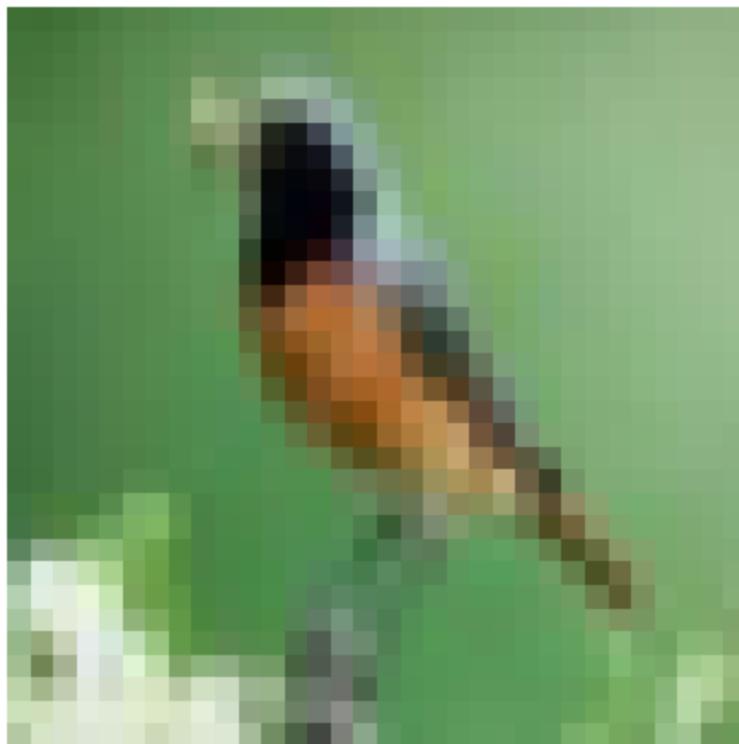
channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

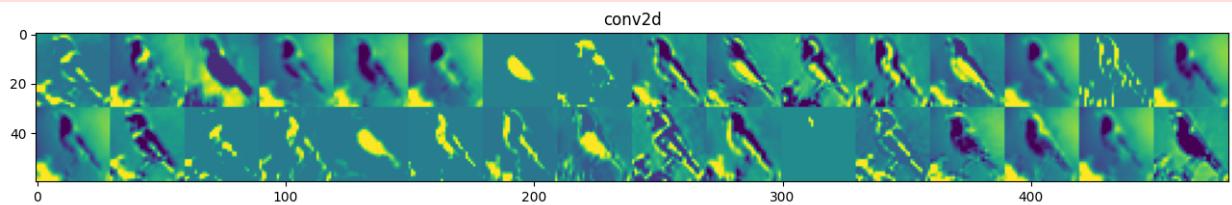
```

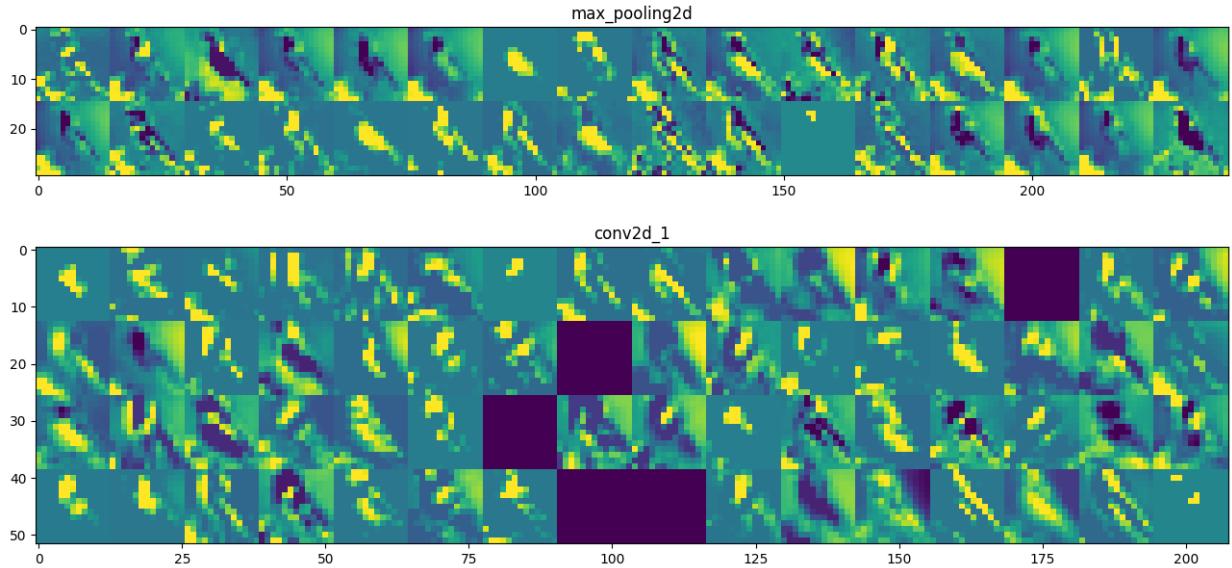


1/1 [=====] - 0s 73ms/step

<ipython-input-267-4848b71b261c>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[184]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

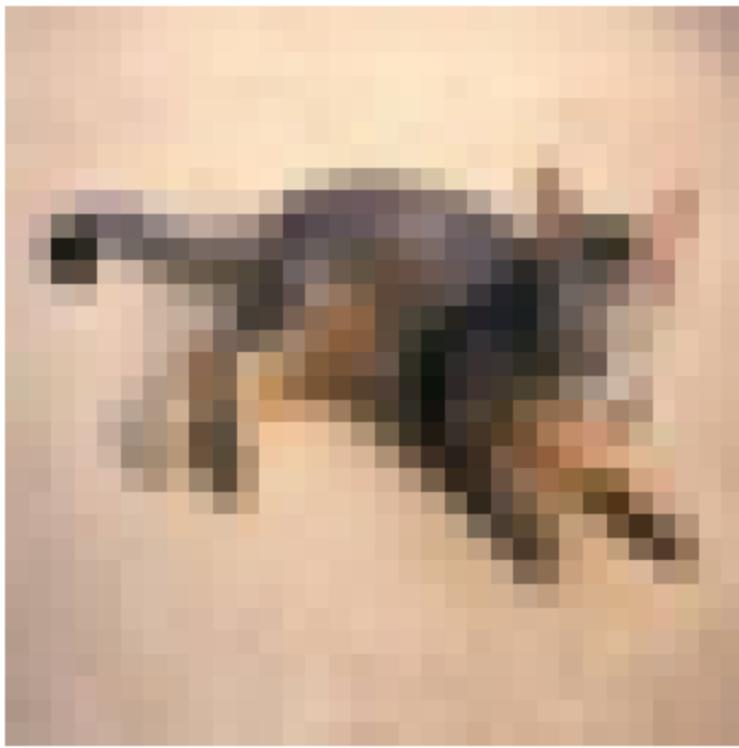
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

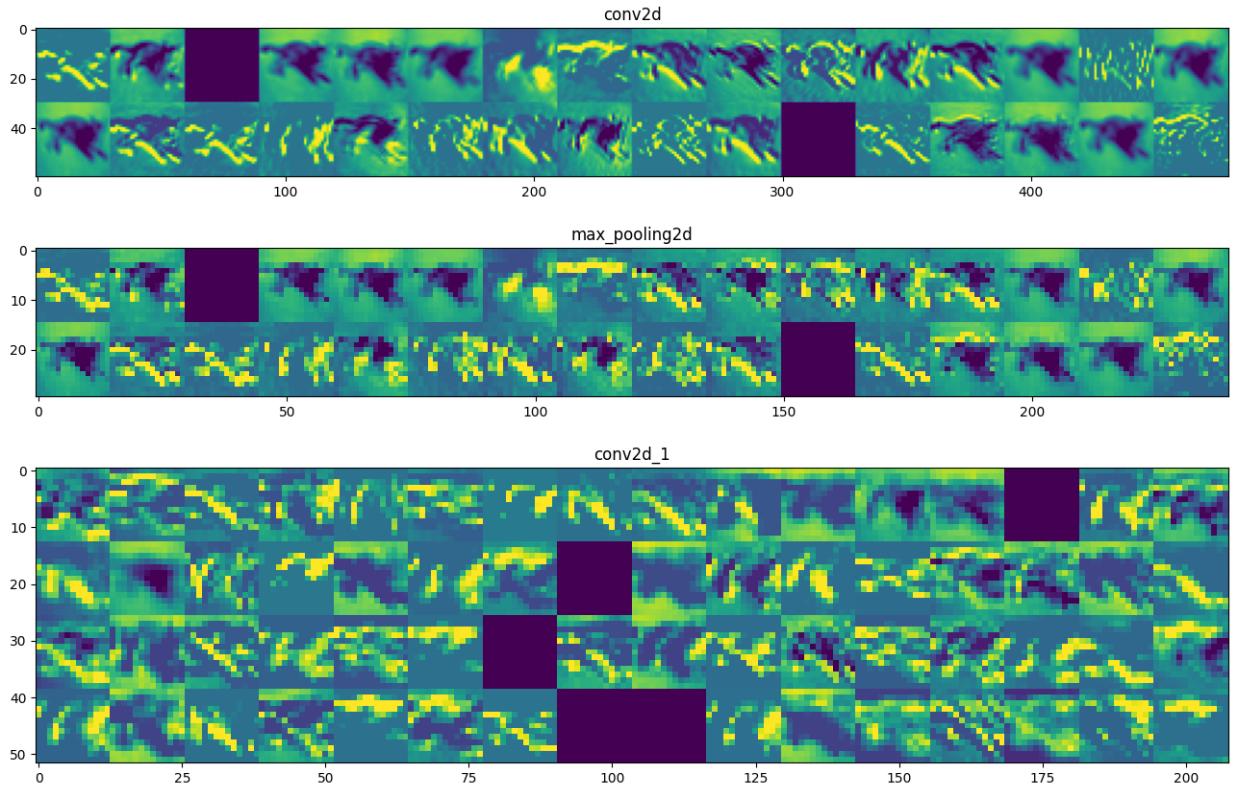
plt.show();
```



```
1/1 [=====] - 0s 78ms/step
```

```
<ipython-input-268-3bac8bdd9965>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[159]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

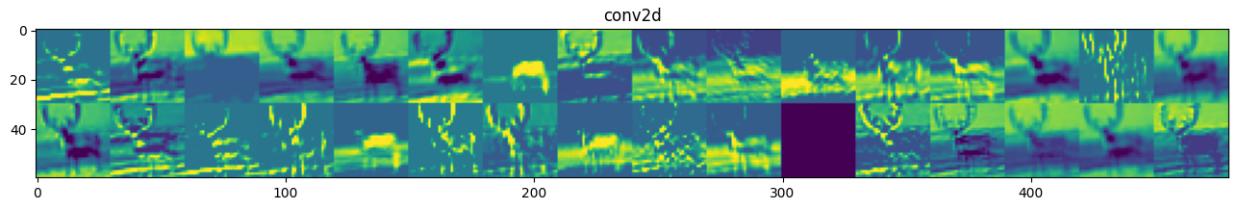
```

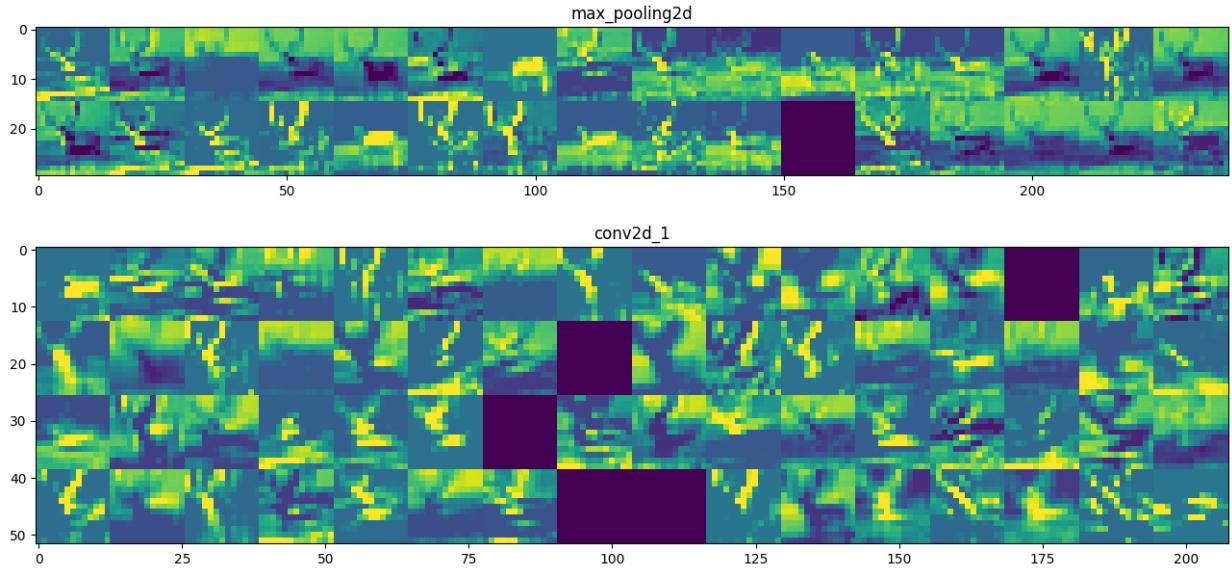


1/1 [=====] - 0s 79ms/step

<ipython-input-269-a52112e96c59>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[24]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

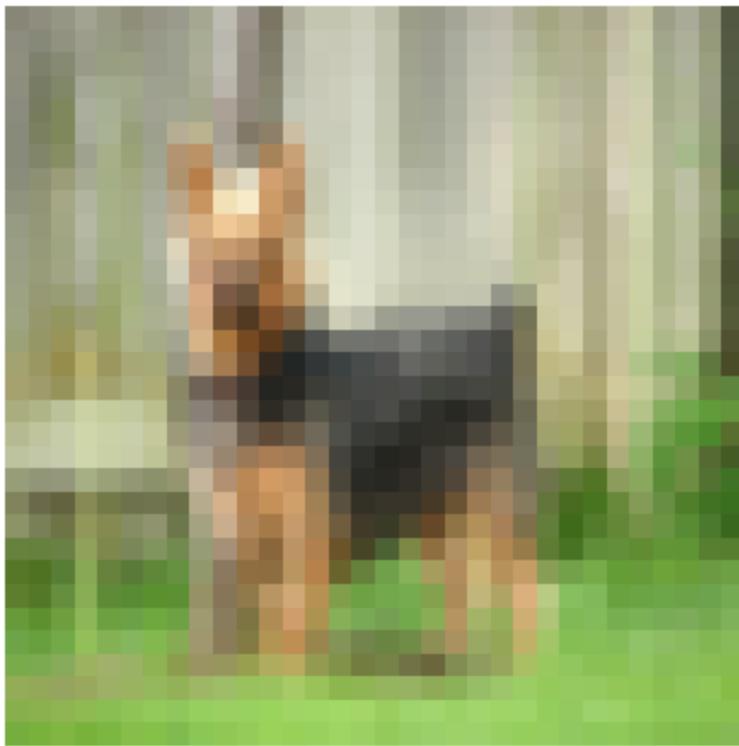
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

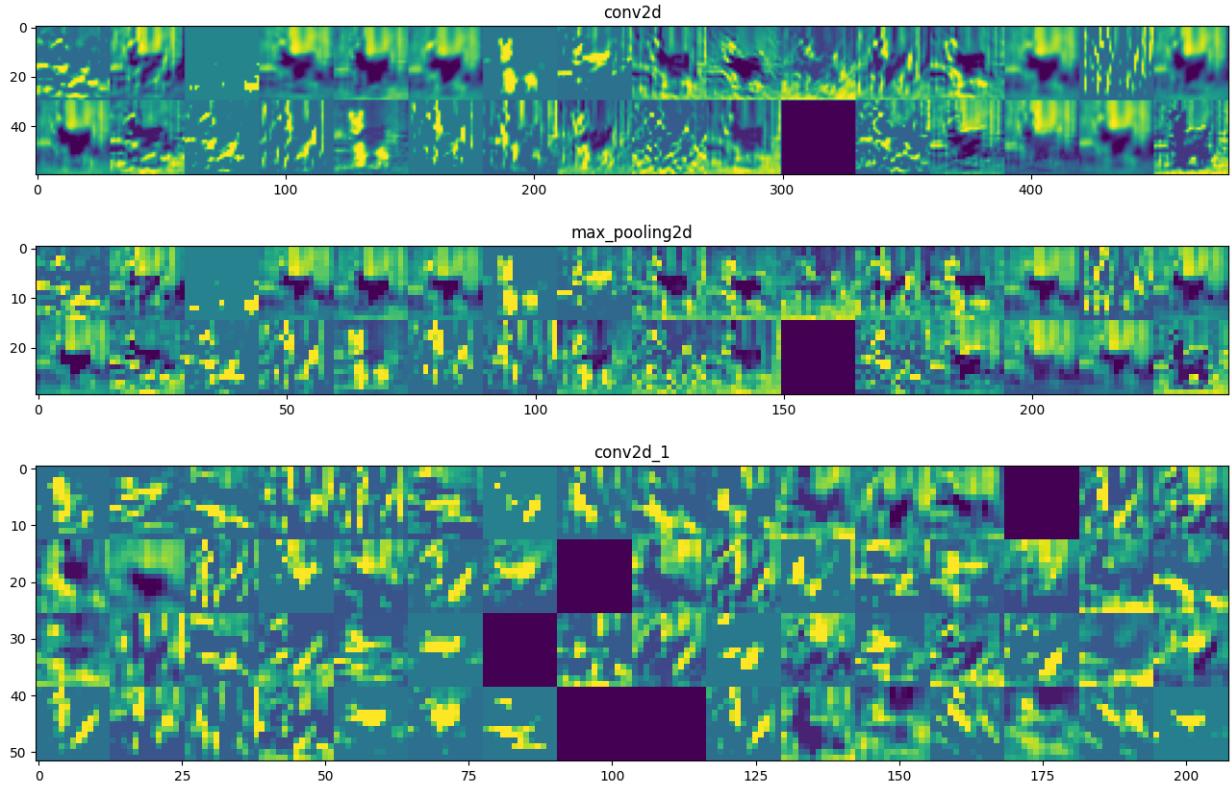
plt.show();
```



```
1/1 [=====] - 0s 87ms/step
```

```
<ipython-input-270-edee34e9d17b>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[152]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

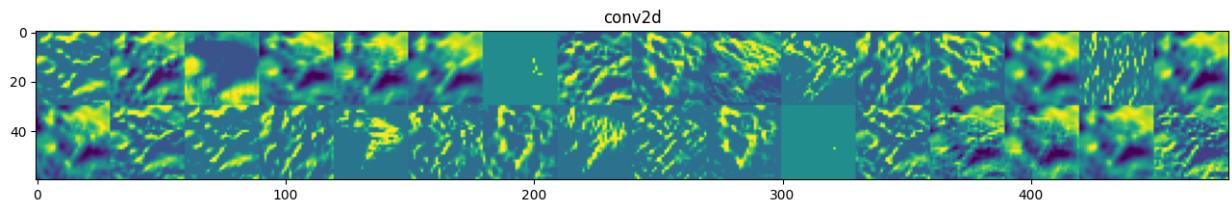
```

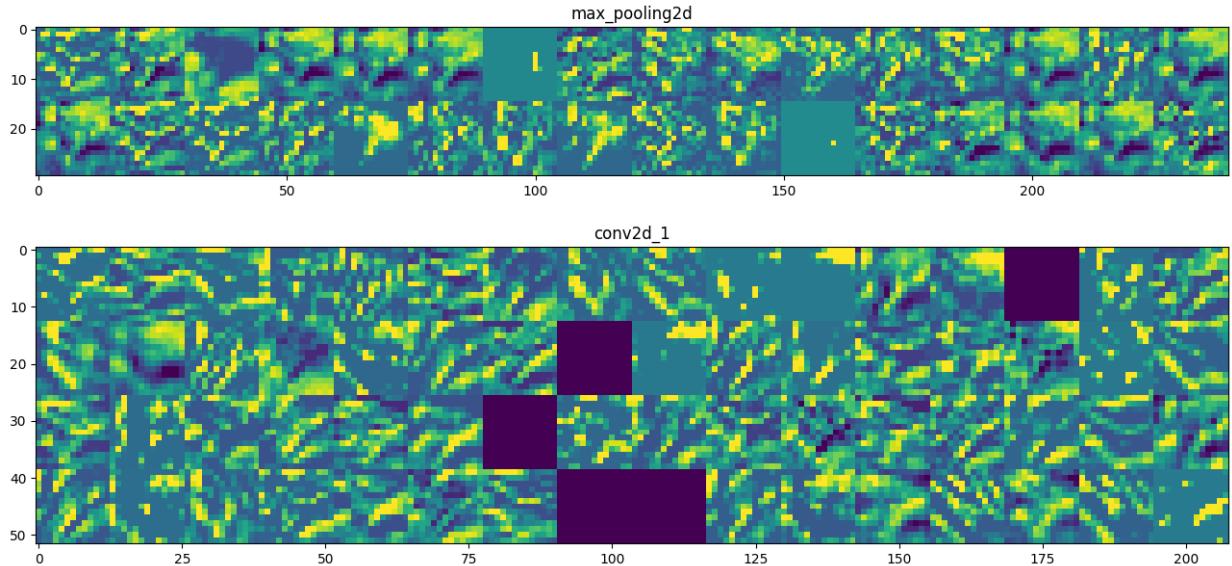


1/1 [=====] - 0s 76ms/step

<ipython-input-271-0ef3f9c61bfb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[2004]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

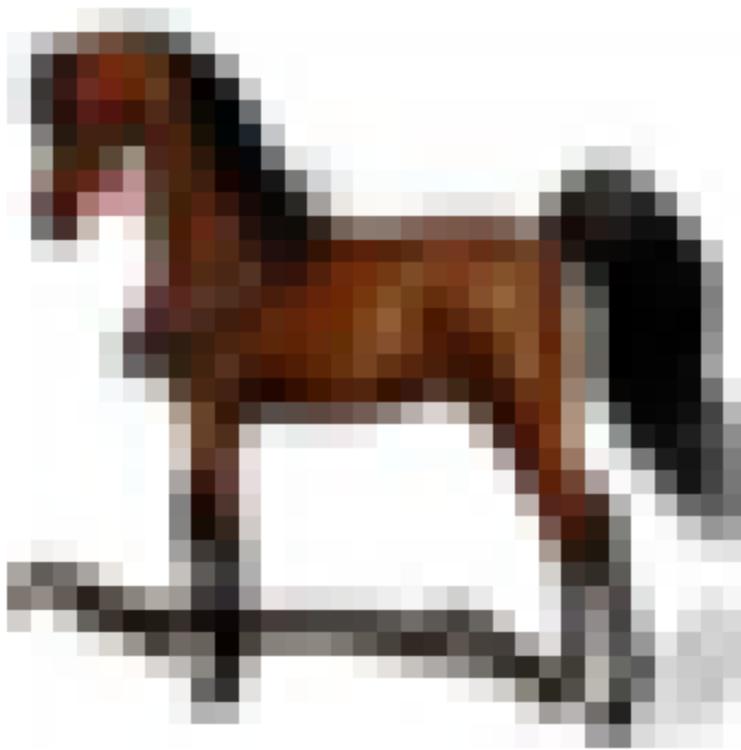
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

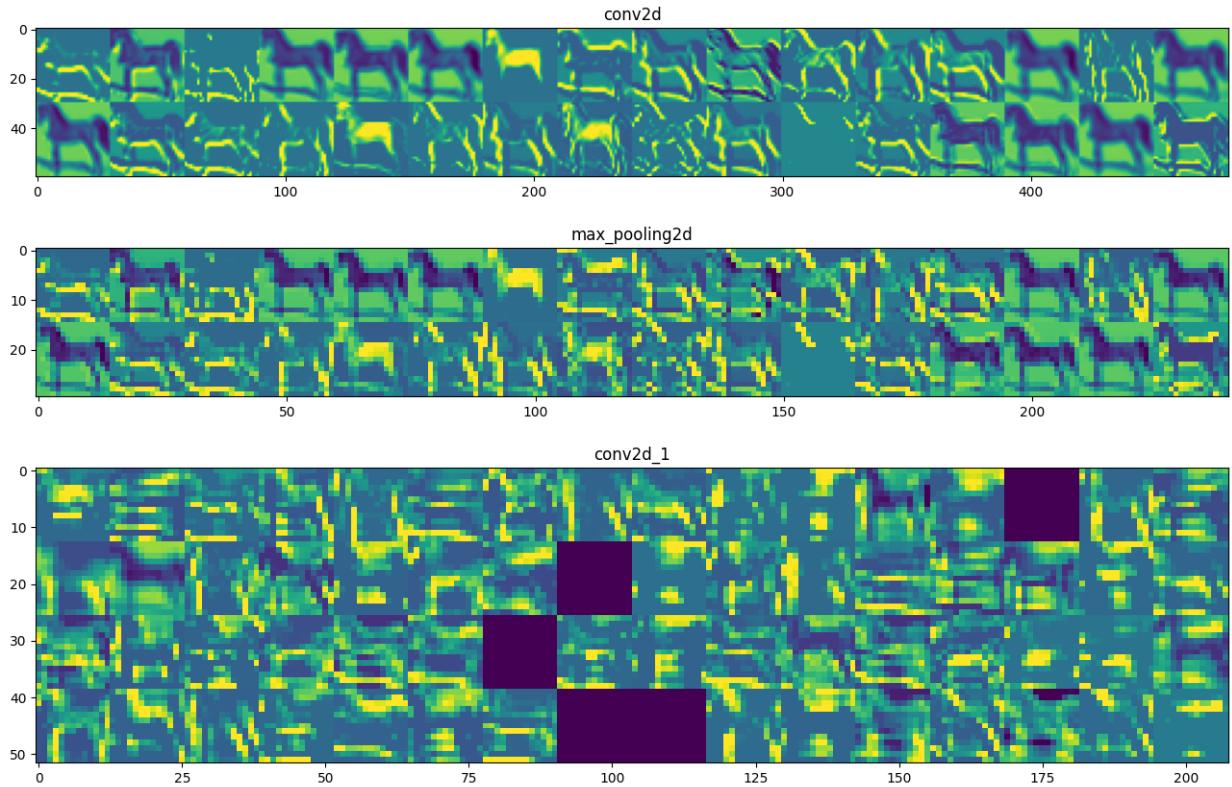
plt.show();
```



```
1/1 [=====] - 0s 87ms/step
```

```
<ipython-input-272-5c55fce06c66>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[185]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
```

```

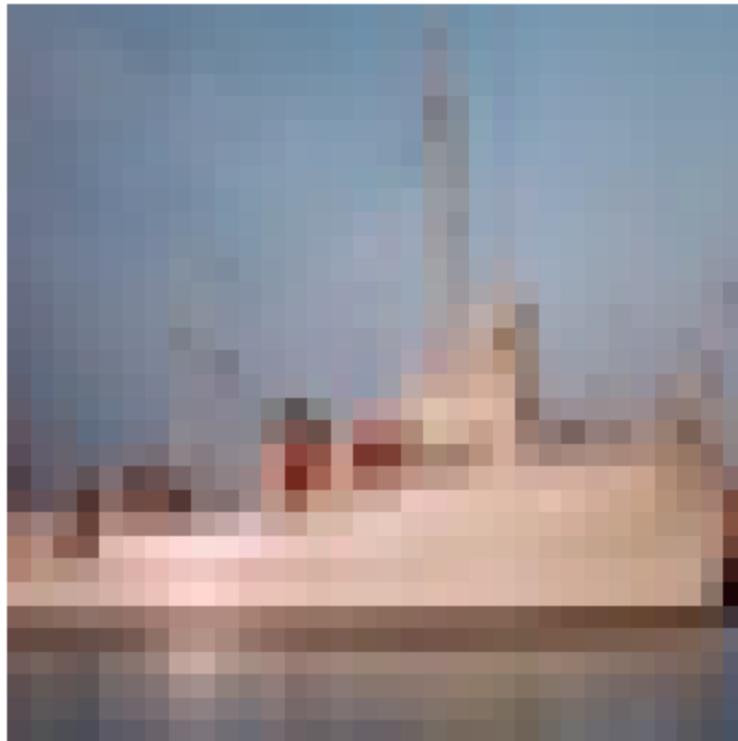
channel_image = layer_activation[0,
                                :, :,
                                col * images_per_row + row]

# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

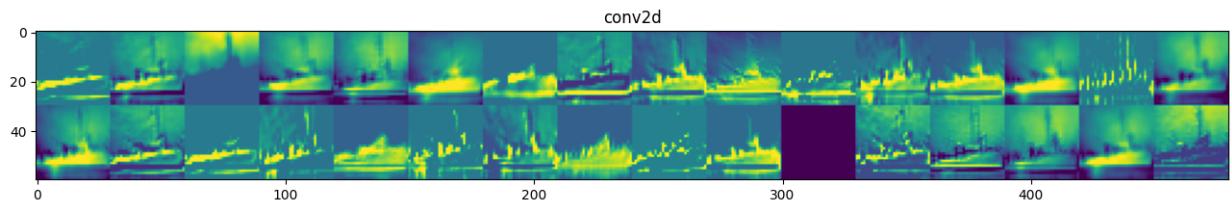
```

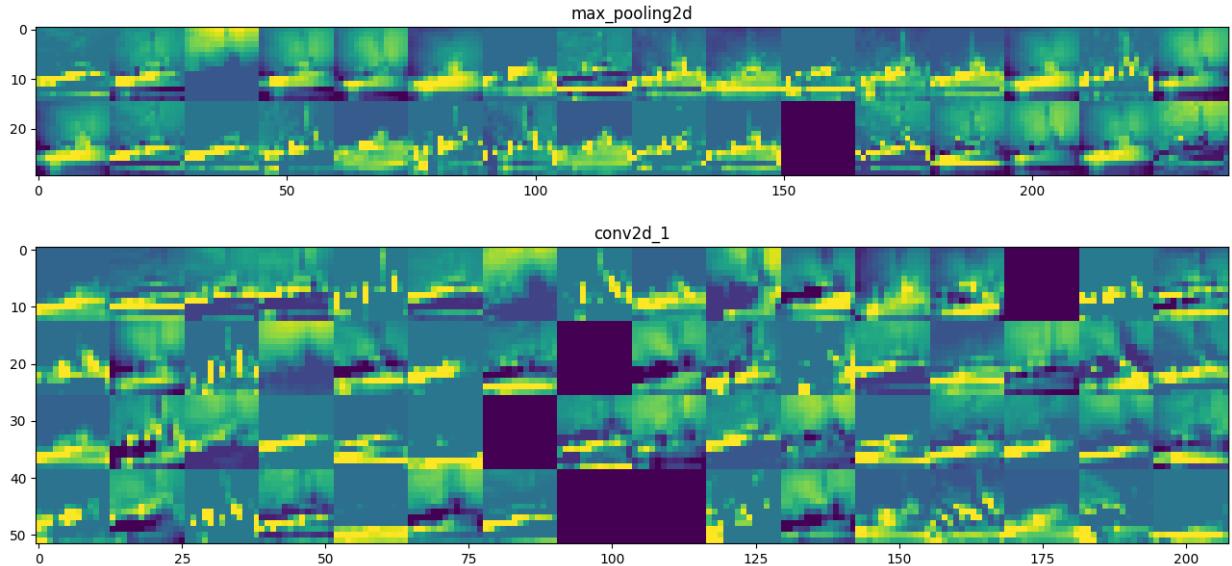


1/1 [=====] - 0s 86ms/step

<ipython-input-273-f532589aadeb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[133]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

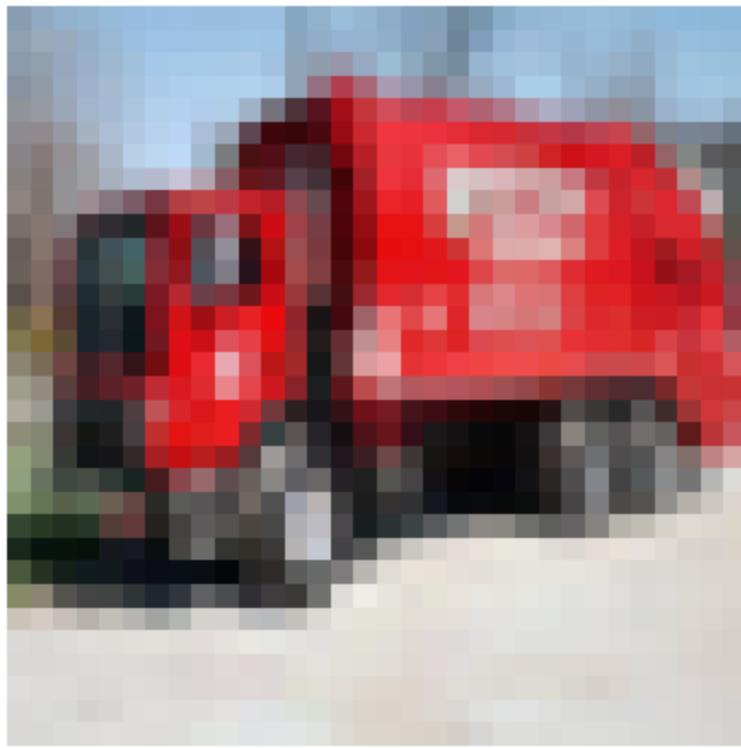
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

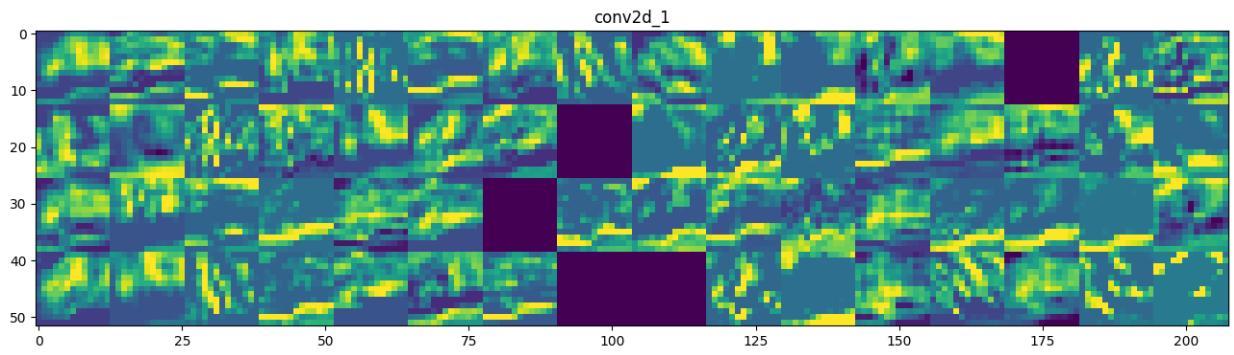
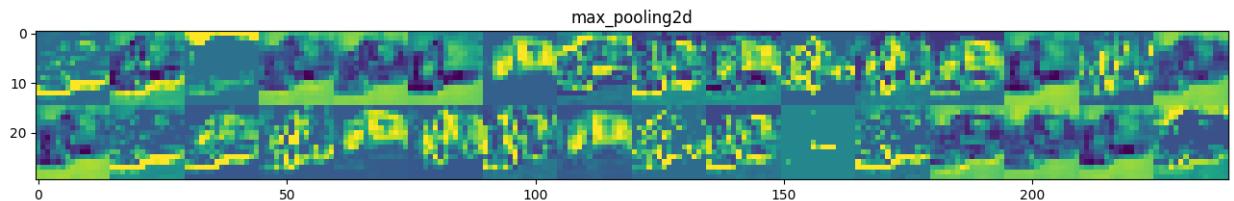
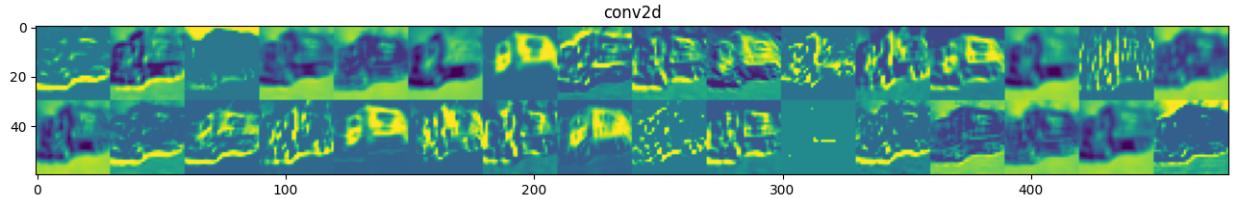
plt.show();
```



```
1/1 [=====] - 0s 130ms/step
```

```
<ipython-input-274-8e6847644768>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



In []:

9) Model 8 - CNN with 3 Max Pooling / Hidden Layers and Early Stopping, Batch Normalization, and Dropout Regularization

8.1) Build The Model

We use a Sequential class defined in Keras to create our model.

```
In [ ]: k.clear_session()
model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu),
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Flatten(),
    Dense(units=256,activation=tf.nn.softmax),
    BatchNormalization(),
    Dropout(0.3),
    Dense(units=10, activation=tf.nn.softmax)
])
```

In []:

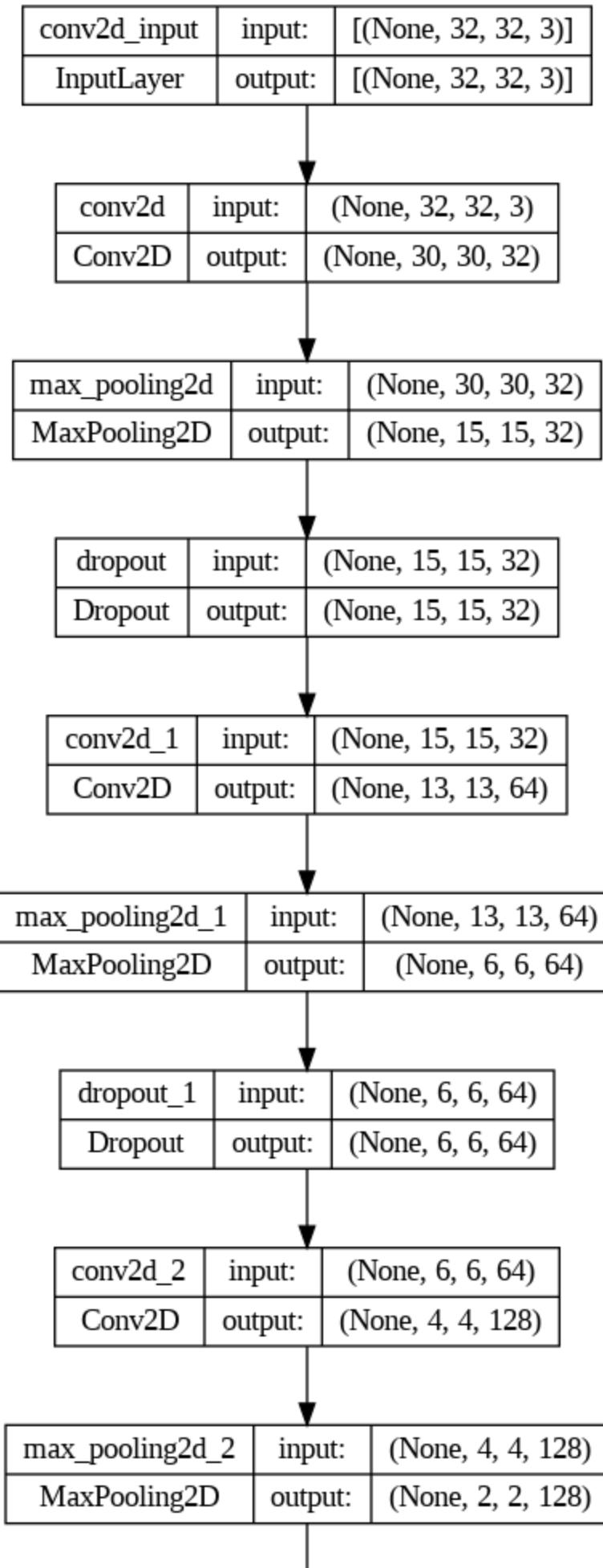
```
model.summary()
```

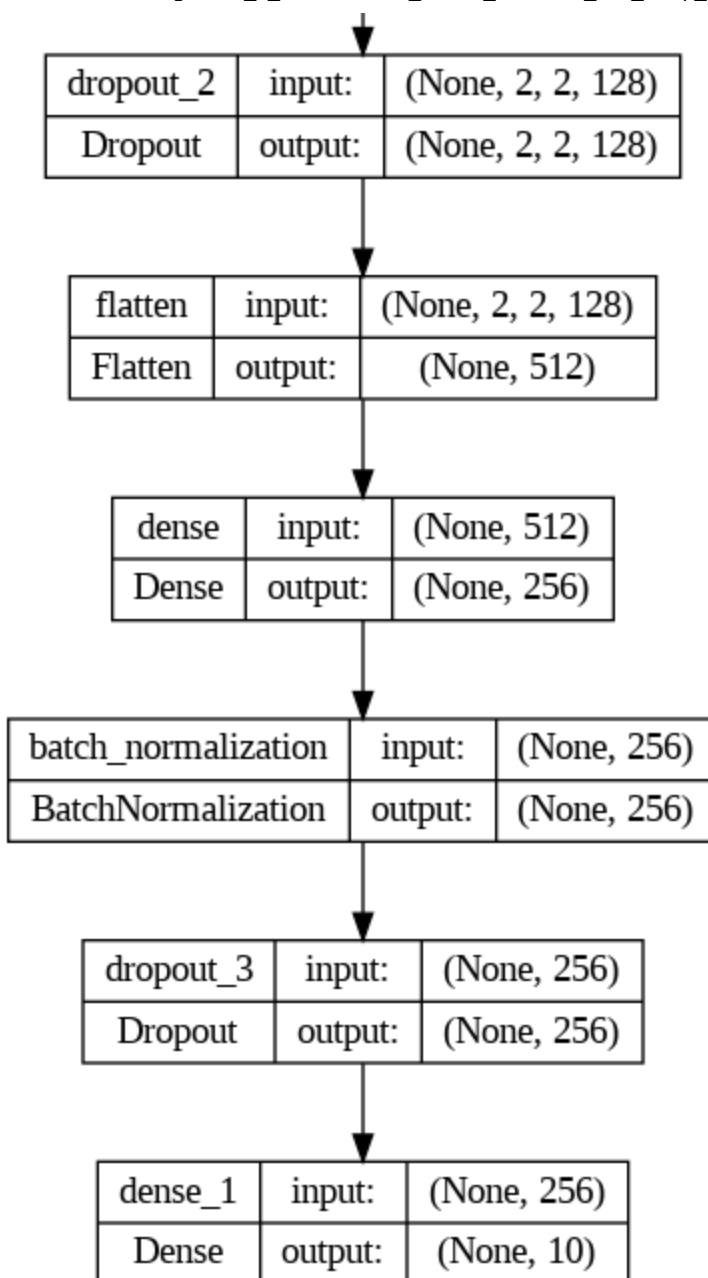
Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
dropout (Dropout)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
dropout_1 (Dropout)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
dropout_2 (Dropout)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 256)	131328
batch_normalization (Batch Normalization)	(None, 256)	1024
dropout_3 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2570
<hr/>		
Total params: 228170 (891.29 KB)		
Trainable params: 227658 (889.29 KB)		
Non-trainable params: 512 (2.00 KB)		

```
In [ ]: tf.keras.utils.plot_model(model, "CIFAR10.png", show_shapes=True)
```

Out[]:





Let's now compile and train the model.

tf.keras.losses.SparseCategoricalCrossentropy

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Module: tf.keras.callbacks

tf.keras.callbacks.EarlyStopping

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

tf.keras.callbacks.ModelCheckpointhttps://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

```
In [ ]: model.compile(optimizer='adam',
                      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
                      metrics=['accuracy'])
```

```
In [ ]: history = model.fit(x_train_norm
                           ,y_train_split
                           ,epochs=40
                           ,batch_size=500
                           ,validation_data=(x_valid_norm, y_valid_split)
                           ,callbacks=[
                           tf.keras.callbacks.ModelCheckpoint("Model_8", save_best_only=True,
                           ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=10)
                           ])
```

```
Epoch 1/40
90/90 [=====] - 71s 748ms/step - loss: 2.0004 - accuracy: 0.
2563 - val_loss: 2.2668 - val_accuracy: 0.1088
Epoch 2/40
90/90 [=====] - 64s 709ms/step - loss: 1.6442 - accuracy: 0.
3974 - val_loss: 2.2069 - val_accuracy: 0.1082
Epoch 3/40
90/90 [=====] - 45s 497ms/step - loss: 1.4799 - accuracy: 0.
4604 - val_loss: 2.1006 - val_accuracy: 0.1842
Epoch 4/40
90/90 [=====] - 40s 443ms/step - loss: 1.3846 - accuracy: 0.
4999 - val_loss: 1.9531 - val_accuracy: 0.3574
Epoch 5/40
90/90 [=====] - 41s 457ms/step - loss: 1.3176 - accuracy: 0.
5246 - val_loss: 1.7838 - val_accuracy: 0.4412
Epoch 6/40
90/90 [=====] - 41s 450ms/step - loss: 1.2626 - accuracy: 0.
5495 - val_loss: 1.4890 - val_accuracy: 0.5644
Epoch 7/40
90/90 [=====] - 40s 446ms/step - loss: 1.2078 - accuracy: 0.
5701 - val_loss: 1.2976 - val_accuracy: 0.5810
Epoch 8/40
90/90 [=====] - 40s 450ms/step - loss: 1.1607 - accuracy: 0.
5875 - val_loss: 1.1280 - val_accuracy: 0.6210
Epoch 9/40
90/90 [=====] - 40s 450ms/step - loss: 1.1293 - accuracy: 0.
5980 - val_loss: 1.0390 - val_accuracy: 0.6350
Epoch 10/40
90/90 [=====] - 40s 443ms/step - loss: 1.0924 - accuracy: 0.
6147 - val_loss: 1.0123 - val_accuracy: 0.6422
Epoch 11/40
90/90 [=====] - 40s 446ms/step - loss: 1.0692 - accuracy: 0.
6230 - val_loss: 0.9828 - val_accuracy: 0.6484
Epoch 12/40
90/90 [=====] - 40s 442ms/step - loss: 1.0356 - accuracy: 0.
6327 - val_loss: 0.9649 - val_accuracy: 0.6606
Epoch 13/40
90/90 [=====] - 40s 449ms/step - loss: 1.0179 - accuracy: 0.
6419 - val_loss: 0.9090 - val_accuracy: 0.6794
Epoch 14/40
90/90 [=====] - 42s 467ms/step - loss: 0.9923 - accuracy: 0.
6481 - val_loss: 0.9012 - val_accuracy: 0.6776
Epoch 15/40
90/90 [=====] - 40s 444ms/step - loss: 0.9730 - accuracy: 0.
6566 - val_loss: 0.8700 - val_accuracy: 0.6950
Epoch 16/40
90/90 [=====] - 40s 438ms/step - loss: 0.9528 - accuracy: 0.
6645 - val_loss: 0.9705 - val_accuracy: 0.6574
Epoch 17/40
90/90 [=====] - 40s 441ms/step - loss: 0.9386 - accuracy: 0.
6704 - val_loss: 0.8576 - val_accuracy: 0.6946
Epoch 18/40
90/90 [=====] - 40s 447ms/step - loss: 0.9203 - accuracy: 0.
6748 - val_loss: 0.8286 - val_accuracy: 0.7066
Epoch 19/40
90/90 [=====] - 39s 431ms/step - loss: 0.9030 - accuracy: 0.
6815 - val_loss: 0.8514 - val_accuracy: 0.6960
Epoch 20/40
90/90 [=====] - 39s 428ms/step - loss: 0.8891 - accuracy: 0.
6870 - val_loss: 0.8533 - val_accuracy: 0.6912
```

```
Epoch 21/40
90/90 [=====] - 39s 439ms/step - loss: 0.8760 - accuracy: 0.
6922 - val_loss: 0.7865 - val_accuracy: 0.7194
Epoch 22/40
90/90 [=====] - 38s 428ms/step - loss: 0.8606 - accuracy: 0.
6965 - val_loss: 0.8380 - val_accuracy: 0.7054
Epoch 23/40
90/90 [=====] - 38s 428ms/step - loss: 0.8540 - accuracy: 0.
6990 - val_loss: 0.8485 - val_accuracy: 0.6964
Epoch 24/40
90/90 [=====] - 39s 438ms/step - loss: 0.8453 - accuracy: 0.
7031 - val_loss: 0.7574 - val_accuracy: 0.7324
Epoch 25/40
90/90 [=====] - 40s 446ms/step - loss: 0.8264 - accuracy: 0.
7116 - val_loss: 0.7544 - val_accuracy: 0.7314
Epoch 26/40
90/90 [=====] - 39s 430ms/step - loss: 0.8232 - accuracy: 0.
7120 - val_loss: 0.7582 - val_accuracy: 0.7276
Epoch 27/40
90/90 [=====] - 40s 444ms/step - loss: 0.8117 - accuracy: 0.
7152 - val_loss: 0.7510 - val_accuracy: 0.7340
Epoch 28/40
90/90 [=====] - 40s 444ms/step - loss: 0.7997 - accuracy: 0.
7181 - val_loss: 0.7436 - val_accuracy: 0.7340
Epoch 29/40
90/90 [=====] - 40s 441ms/step - loss: 0.7917 - accuracy: 0.
7216 - val_loss: 0.7114 - val_accuracy: 0.7512
Epoch 30/40
90/90 [=====] - 39s 428ms/step - loss: 0.7827 - accuracy: 0.
7237 - val_loss: 0.7220 - val_accuracy: 0.7444
Epoch 31/40
90/90 [=====] - 39s 430ms/step - loss: 0.7798 - accuracy: 0.
7266 - val_loss: 0.7126 - val_accuracy: 0.7468
Epoch 32/40
90/90 [=====] - 40s 442ms/step - loss: 0.7682 - accuracy: 0.
7285 - val_loss: 0.6987 - val_accuracy: 0.7548
Epoch 33/40
90/90 [=====] - 39s 430ms/step - loss: 0.7646 - accuracy: 0.
7325 - val_loss: 0.7812 - val_accuracy: 0.7230
Epoch 34/40
90/90 [=====] - 39s 429ms/step - loss: 0.7600 - accuracy: 0.
7336 - val_loss: 0.7098 - val_accuracy: 0.7462
Epoch 35/40
90/90 [=====] - 39s 439ms/step - loss: 0.7509 - accuracy: 0.
7363 - val_loss: 0.6797 - val_accuracy: 0.7610
Epoch 36/40
90/90 [=====] - 39s 430ms/step - loss: 0.7470 - accuracy: 0.
7368 - val_loss: 0.7118 - val_accuracy: 0.7444
Epoch 37/40
90/90 [=====] - 38s 428ms/step - loss: 0.7457 - accuracy: 0.
7393 - val_loss: 0.7199 - val_accuracy: 0.7426
Epoch 38/40
90/90 [=====] - 39s 440ms/step - loss: 0.7392 - accuracy: 0.
7406 - val_loss: 0.7162 - val_accuracy: 0.7476
```

8.2) Evaluate Model Performance

```
In [ ]: model = tf.keras.models.load_model("Model_8")
print(f"Training accuracy: {model.evaluate(x_train_norm, y_train_split)[1]:.3f}")
```

```
print(f"Validation accuracy: {model.evaluate(x_valid_norm, y_valid_split)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_norm, y_test)[1]:.3f}")

1407/1407 [=====] - 14s 10ms/step - loss: 0.5345 - accuracy: 0.8188
Training accuracy: 0.819
157/157 [=====] - 1s 9ms/step - loss: 0.6797 - accuracy: 0.7610
Validation accuracy: 0.761
313/313 [=====] - 4s 12ms/step - loss: 0.7051 - accuracy: 0.7550
Test accuracy: 0.755
```

In []: `preds = model.predict(x_test_norm)`
`print('shape of preds: ', preds.shape)`

```
313/313 [=====] - 3s 9ms/step
shape of preds: (10000, 10)
```

In []: `history_dict = history.history`
`history_dict.keys()`

Out[]: `dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])`

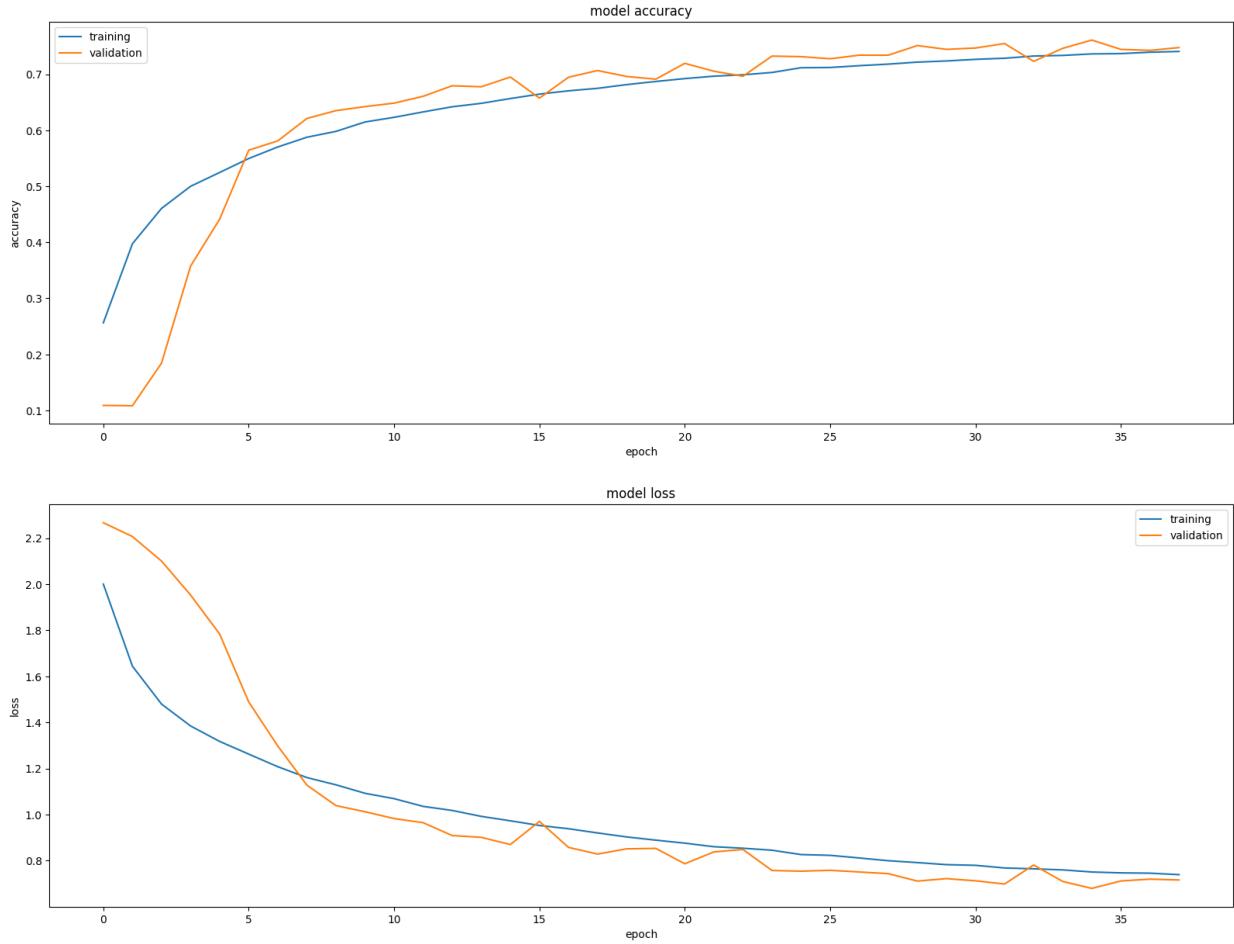
In []: `history_df=pd.DataFrame(history_dict)`
`history_df.tail().round(3)`

Out[]:

	loss	accuracy	val_loss	val_accuracy
33	0.760	0.734	0.710	0.746
34	0.751	0.736	0.680	0.761
35	0.747	0.737	0.712	0.744
36	0.746	0.739	0.720	0.743
37	0.739	0.741	0.716	0.748

In []: `plt.subplots(figsize=(16,12))`
`plt.tight_layout()`
`display_training_curves(history.history['accuracy'], history.history['val_accuracy'],`
`display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',`

<ipython-input-8-353fbbe40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.
`ax = plt.subplot(subplot)`



Let's examine the precision, recall, F1 score, and confusion matrix.

```
In [ ]: pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

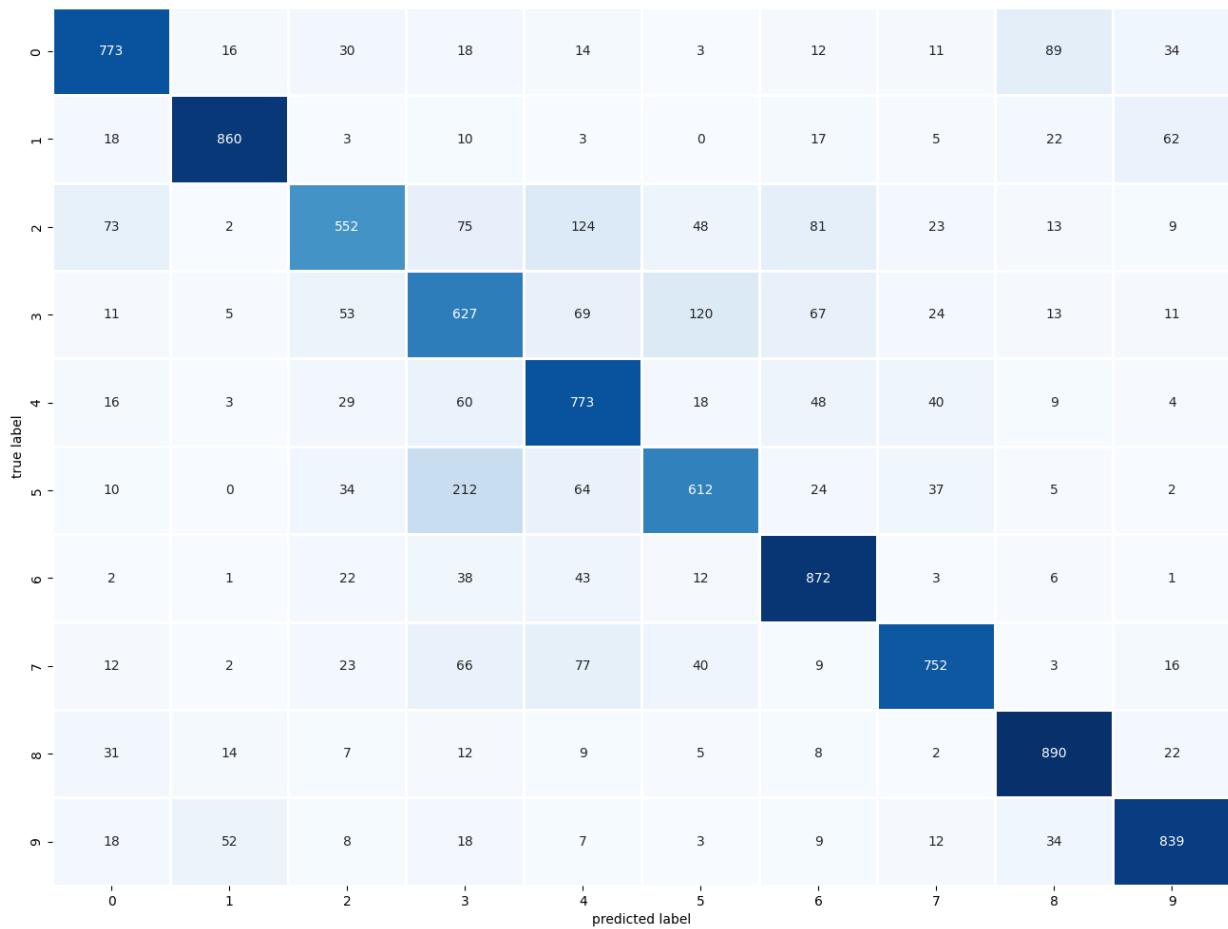
313/313 [=====] - 4s 12ms/step
```

```
In [ ]: print_validation_report(y_test, pred1)
```

Classification Report				
	precision	recall	f1-score	support
0	0.80	0.77	0.79	1000
1	0.90	0.86	0.88	1000
2	0.73	0.55	0.63	1000
3	0.55	0.63	0.59	1000
4	0.65	0.77	0.71	1000
5	0.71	0.61	0.66	1000
6	0.76	0.87	0.81	1000
7	0.83	0.75	0.79	1000
8	0.82	0.89	0.85	1000
9	0.84	0.84	0.84	1000
accuracy			0.76	10000
macro avg	0.76	0.76	0.75	10000
weighted avg	0.76	0.76	0.75	10000

Accuracy Score: 0.755

Root Mean Square Error: 1.9789643756268076

In []: `plot_confusion_matrix(y_test,pred1)`

Load HDF5 Model Format

tf.keras.models.load_modelhttps://www.tensorflow.org/api_docs/python/tf/keras/models/load_modelIn []: `model = tf.keras.models.load_model('Model_8')
preds = model.predict(x_test_norm)
preds.shape`

313/313 [=====] - 3s 10ms/step

Out[]:

Let's examine the predictions for the testing dataset.

In []: `cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)`

```
df = pd.DataFrame(preds[0:20], columns = ['airplane'
                                         , 'automobile'
                                         , 'bird'
                                         , 'cat'
                                         , 'deer'
                                         , 'dog'
                                         , 'frog'
                                         , 'horse']
```

```

        , 'ship'
        , 'truck'])
df.style.format("{:.2%}").background_gradient(cmap=cm)

```

Out[]:

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0	1.30%	0.22%	0.82%	84.36%	0.40%	2.49%	1.77%	0.16%	8.20%	0.28%
1	1.39%	27.55%	0.01%	0.00%	0.00%	0.00%	0.00%	0.01%	70.72%	0.32%
2	3.67%	8.15%	0.24%	0.18%	0.04%	0.07%	0.10%	0.15%	84.36%	3.04%
3	75.56%	2.22%	0.57%	0.33%	0.55%	0.03%	0.04%	0.07%	20.27%	0.36%
4	0.01%	0.01%	1.17%	2.39%	59.90%	0.07%	36.42%	0.01%	0.01%	0.01%
5	0.01%	0.02%	0.14%	3.45%	0.25%	0.69%	95.36%	0.04%	0.01%	0.03%
6	15.10%	38.24%	11.91%	13.88%	0.36%	9.78%	2.32%	1.46%	0.16%	6.81%
7	0.33%	0.01%	9.41%	4.92%	19.49%	0.36%	65.36%	0.04%	0.06%	0.03%
8	0.07%	0.02%	1.33%	81.82%	4.02%	9.01%	1.17%	2.42%	0.05%	0.09%
9	0.39%	80.90%	0.21%	0.03%	0.04%	0.04%	1.66%	0.04%	0.61%	16.09%
10	20.00%	0.11%	7.76%	12.92%	16.38%	11.46%	0.84%	2.74%	27.04%	0.75%
11	0.00%	0.11%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.01%	99.88%
12	0.09%	0.06%	2.85%	24.47%	11.53%	48.97%	5.05%	6.89%	0.04%	0.05%
13	0.00%	0.00%	0.01%	0.05%	0.22%	0.74%	0.00%	98.98%	0.00%	0.01%
14	0.00%	0.75%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.02%	99.22%
15	4.37%	0.28%	1.98%	0.94%	1.57%	0.08%	22.33%	0.01%	68.30%	0.15%
16	0.01%	0.01%	0.36%	27.21%	0.04%	71.26%	0.08%	1.00%	0.01%	0.02%
17	0.35%	0.13%	1.88%	31.33%	6.45%	26.87%	2.60%	28.78%	0.06%	1.55%
18	1.32%	0.45%	0.00%	0.01%	0.00%	0.00%	0.01%	0.00%	95.52%	2.67%
19	0.00%	0.02%	0.05%	0.14%	0.03%	0.01%	99.75%	0.00%	0.00%	0.00%

In []:

```

(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[98]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
    , 'automobile'
    , 'bird'
    , 'cat'
    , 'deer'
    , 'dog'
    , 'frog'
    , 'horse'
    , 'ship'
    , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')

```

```
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

# Display the grid
```

```

scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                    scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

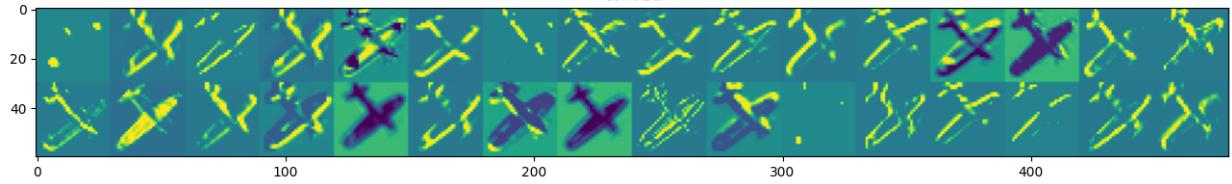
plt.show();

```

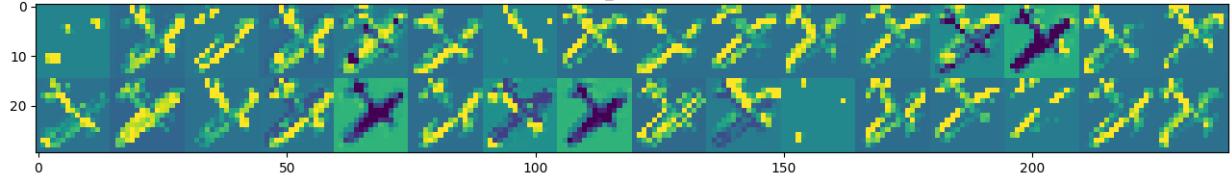


1/1 [=====] - 0s 78ms/step

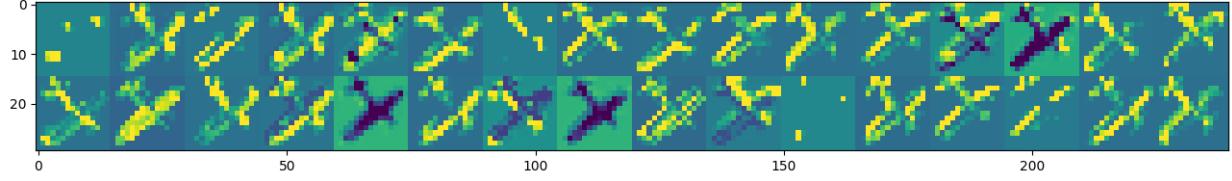
conv2d



max_pooling2d



dropout



In []:

```

(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[122]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

```

```
class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

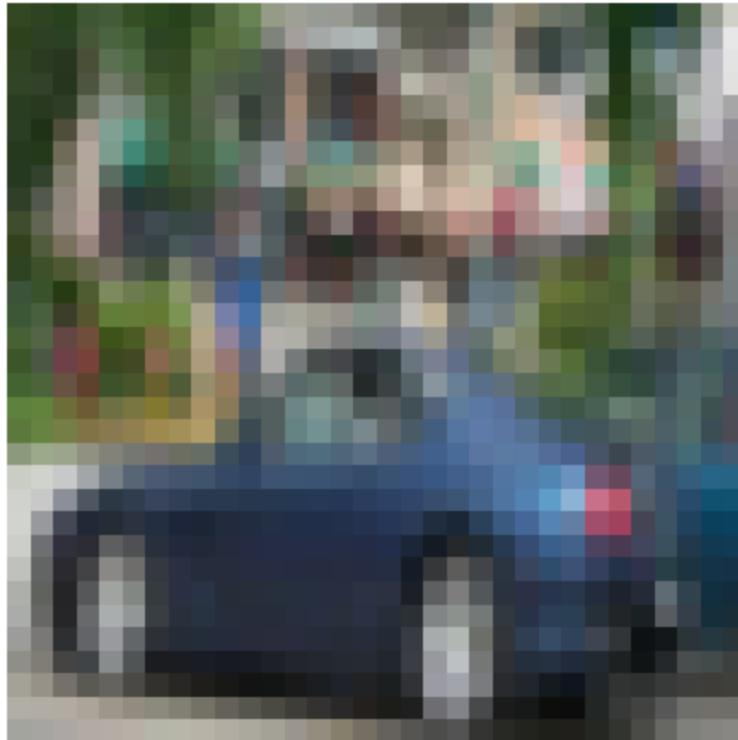
    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
```

```
for row in range(images_per_row):
    channel_image = layer_activation[0,
                                    :, :,
                                    col * images_per_row + row]
    # Post-process the feature to make it visually palatable
    channel_image -= channel_image.mean()
    channel_image /= channel_image.std()
    channel_image *= 64
    channel_image += 128
    channel_image = np.clip(channel_image, 0, 255).astype('uint8')
    display_grid[col * size : (col + 1) * size,
                  row * size : (row + 1) * size] = channel_image

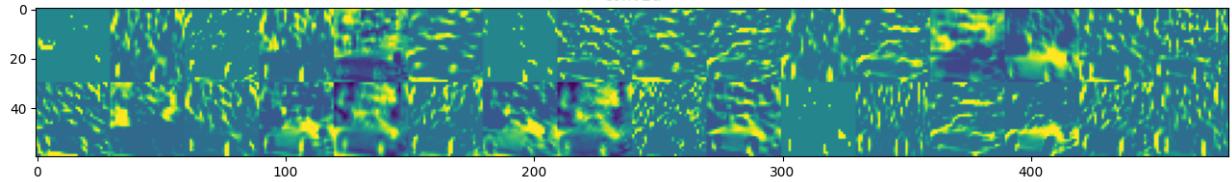
# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

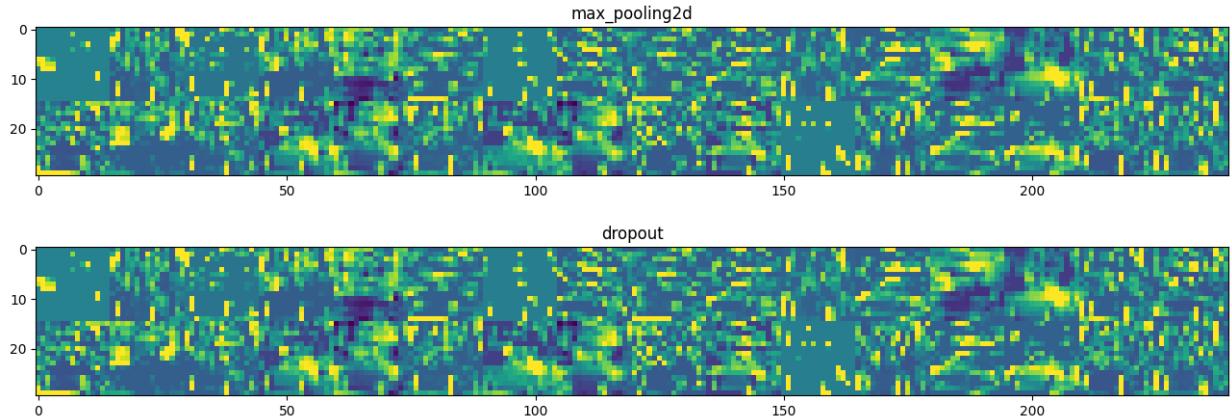
plt.show();
```



1/1 [=====] - 0s 80ms/step

conv2d





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[75]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

These are the names of the Layers, so can have them as part of our plot

```
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

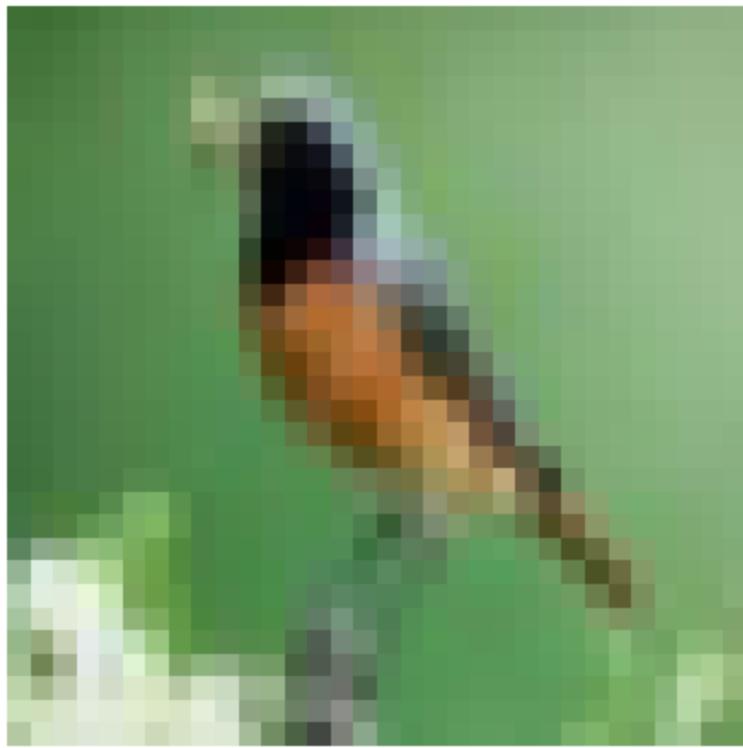
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

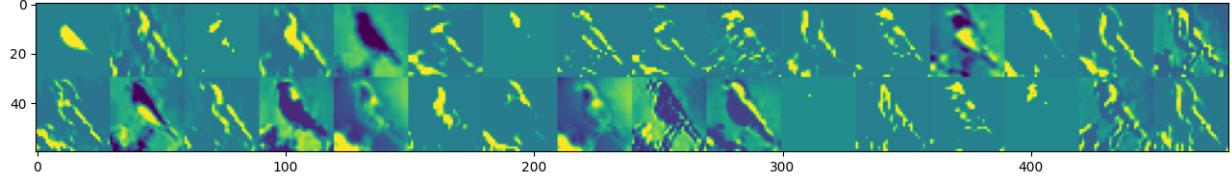
    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

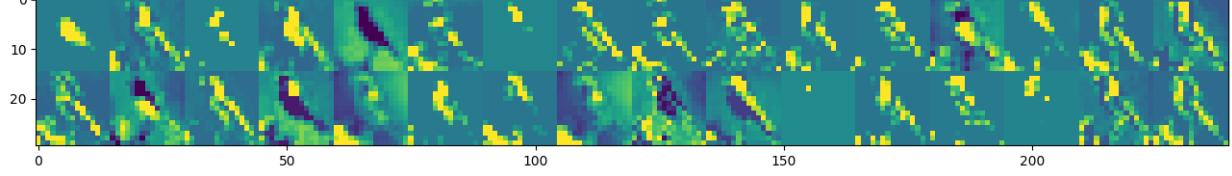


1/1 [=====] - 0s 69ms/step

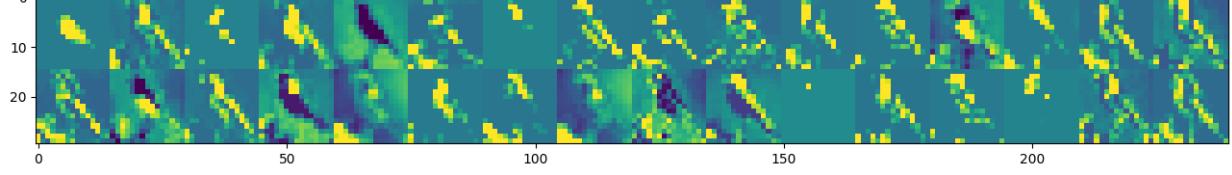
conv2d



max_pooling2d



dropout



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[184]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
,'automobile'
,'bird'
,'cat'
,'deer'
,'dog'
,'frog'
```

```
, 'horse'
, 'ship'
, 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

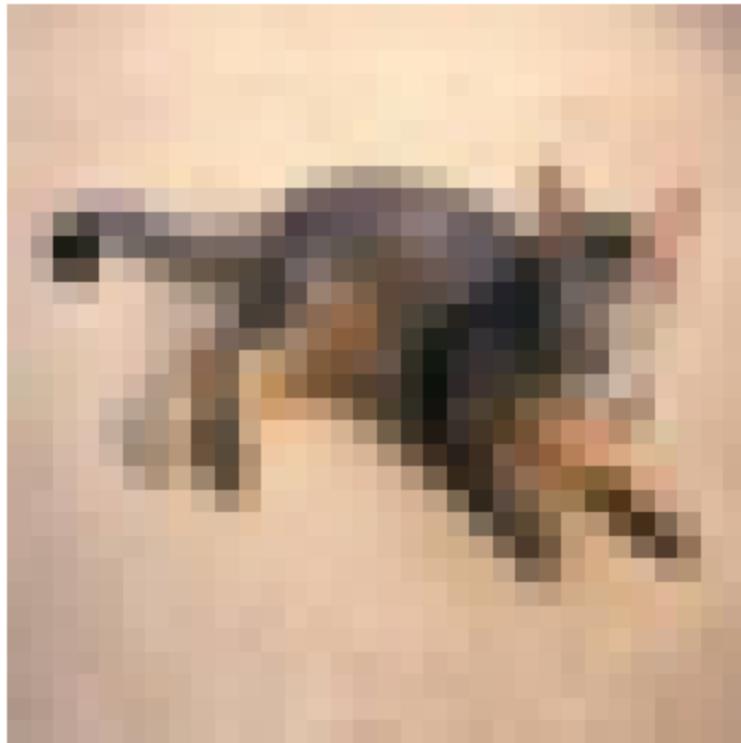
    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
```

```
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

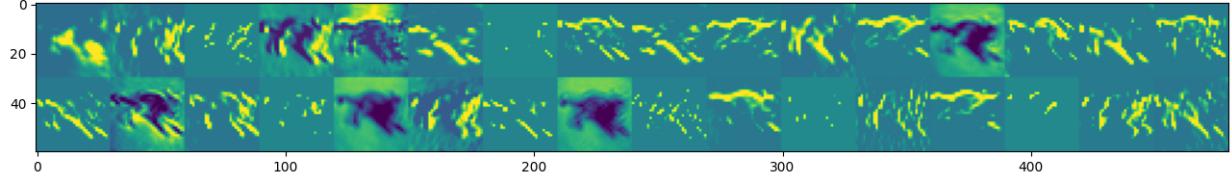
# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

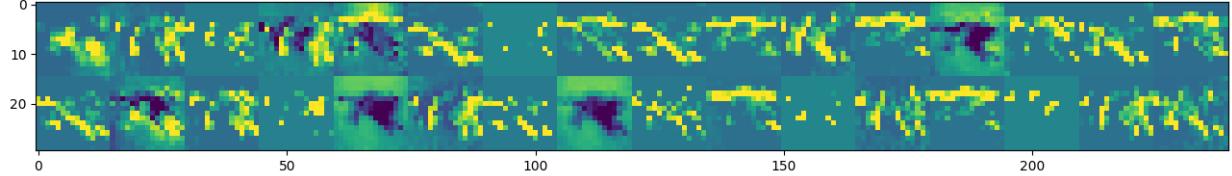


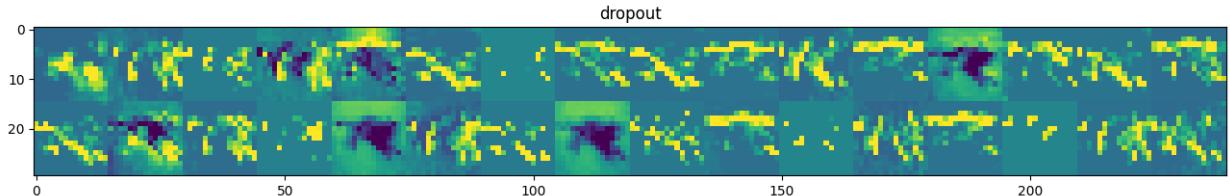
1/1 [=====] - 0s 85ms/step

conv2d



max_pooling2d





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[159]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
```

```
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

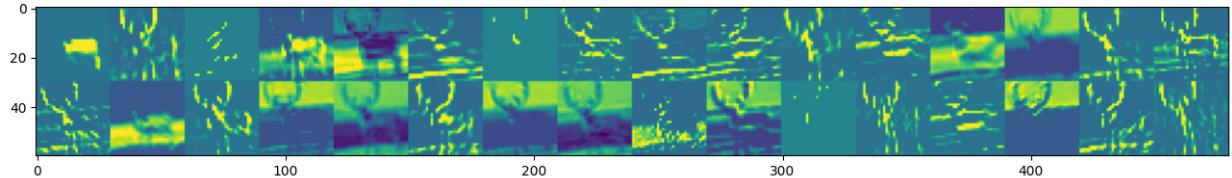
    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

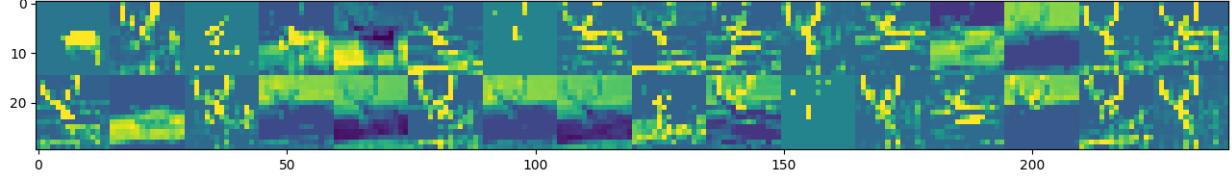


1/1 [=====] - 0s 73ms/step

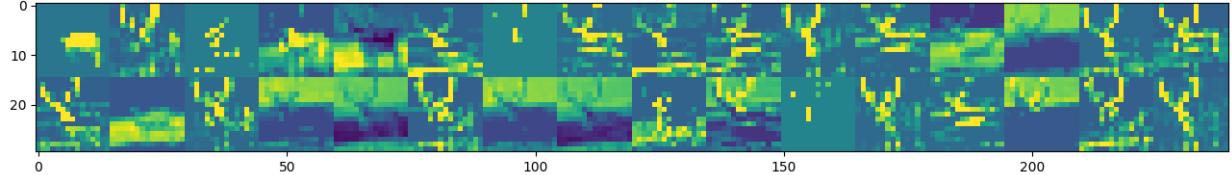
conv2d



max_pooling2d



dropout

In []: `(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()`

```

img = test_images[24]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

```

Extracts the outputs of the top 8 Layers:

```

layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

```

```

activations = activation_model.predict(img_tensor)
len(activations)

```

```

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

```

```
layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

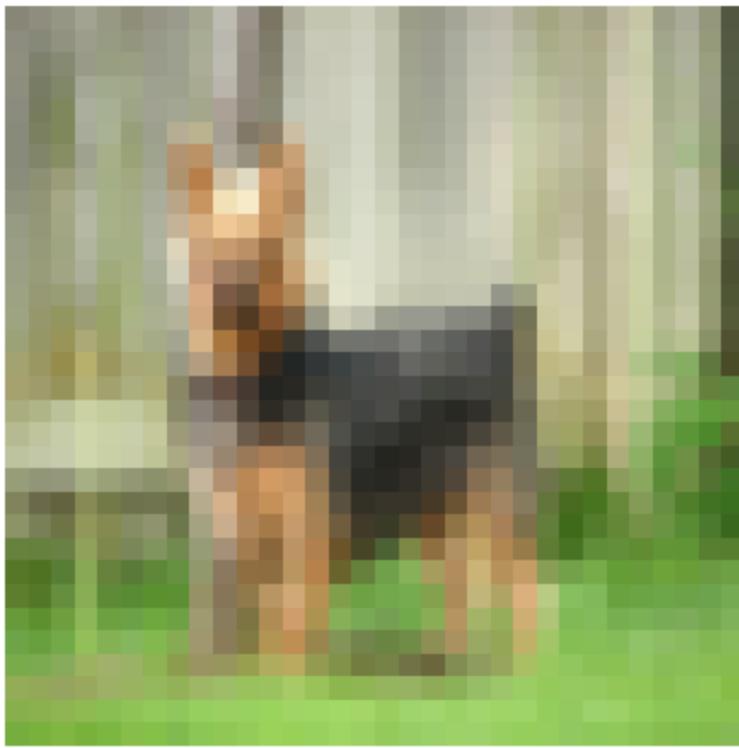
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

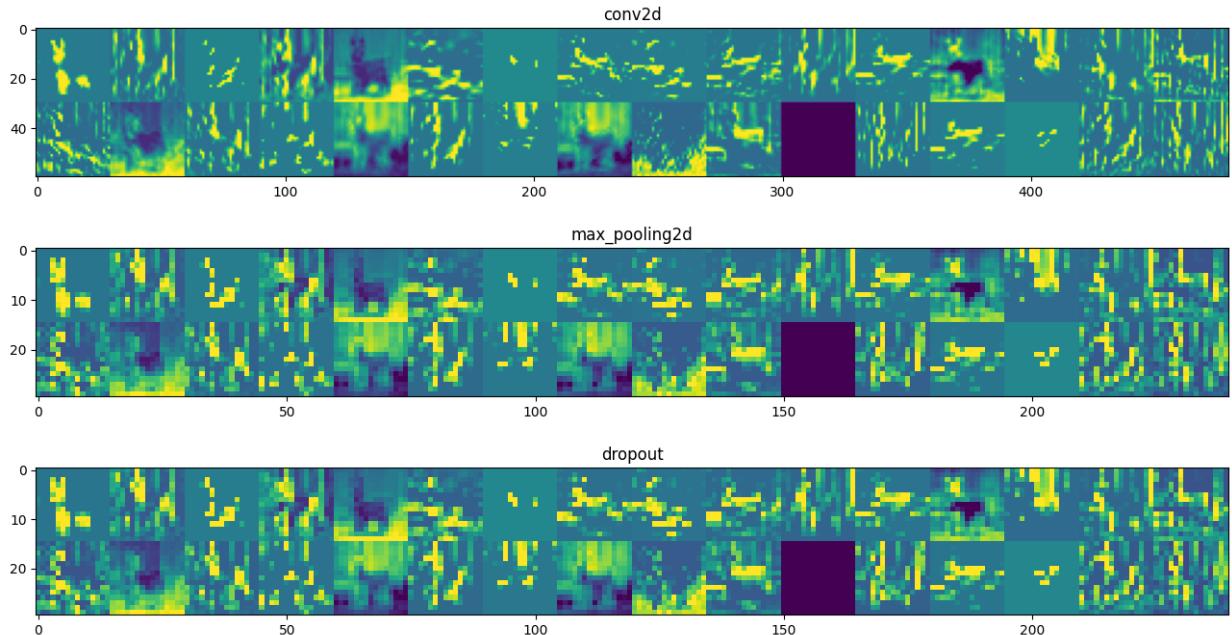
plt.show();
```



```
1/1 [=====] - 0s 71ms/step
```

```
<ipython-input-299-edee34e9d17b>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[152]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat']
```

```

    , 'deer'
    , 'dog'
    , 'frog'
    , 'horse'
    , 'ship'
    , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable

```

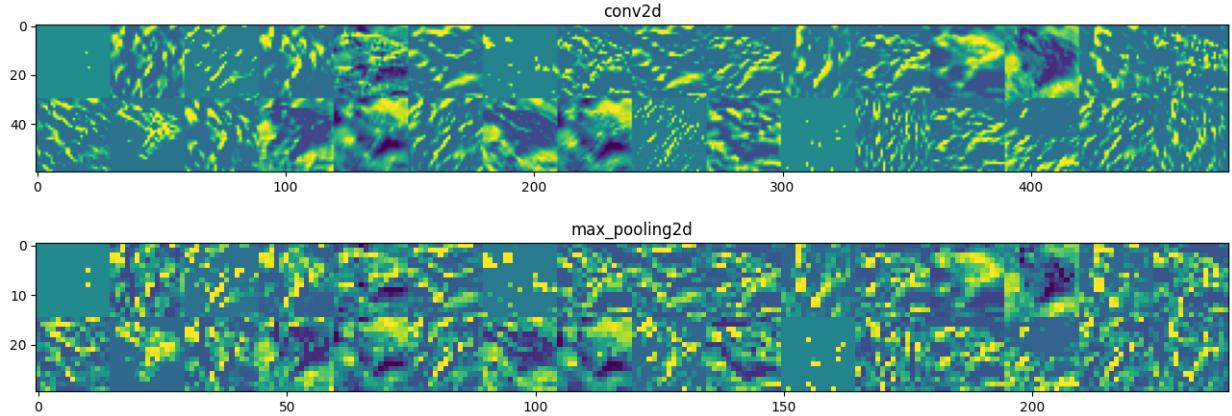
```
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

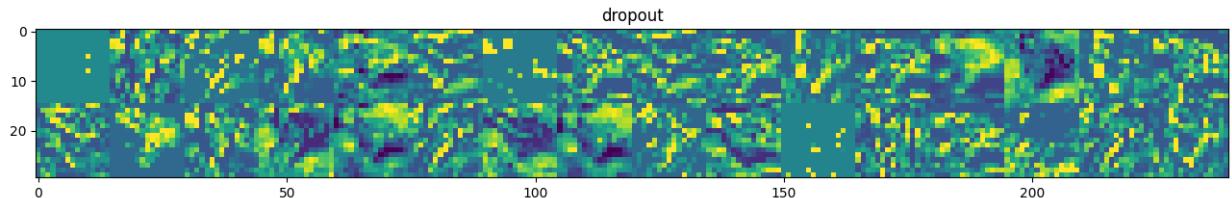
# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```



1/1 [=====] - 0s 71ms/step





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[2004]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
```

```
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

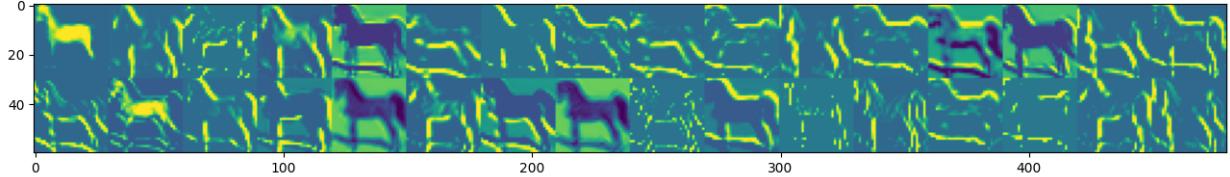
    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

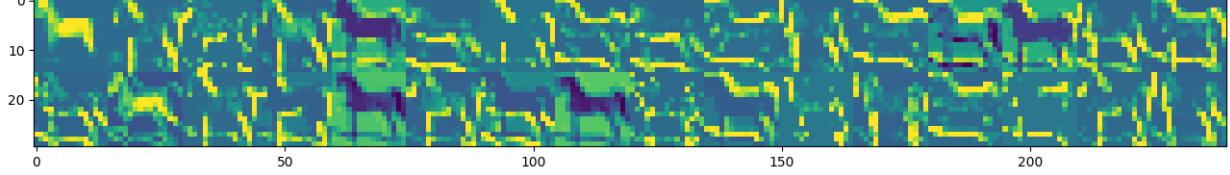


1/1 [=====] - 0s 85ms/step

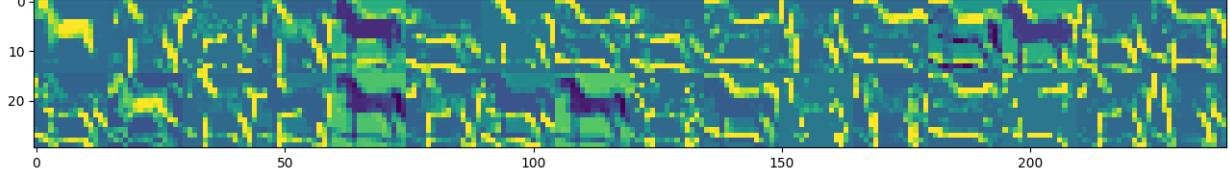
conv2d



max_pooling2d



dropout

In []: `(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()`

```

img = test_images[185]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

```

Extracts the outputs of the top 8 Layers:

```

layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

```

```

activations = activation_model.predict(img_tensor)
len(activations)

```

```

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

```

```
layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

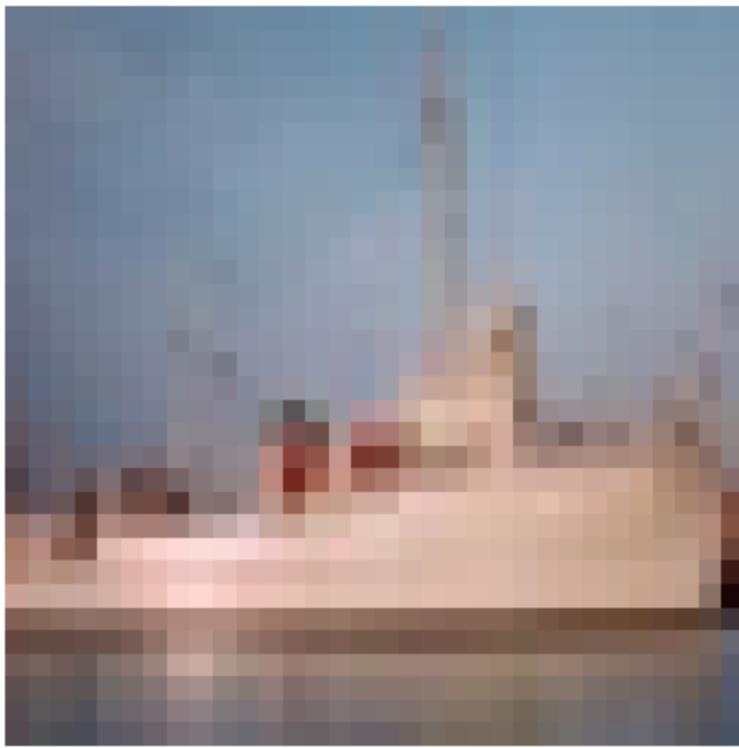
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

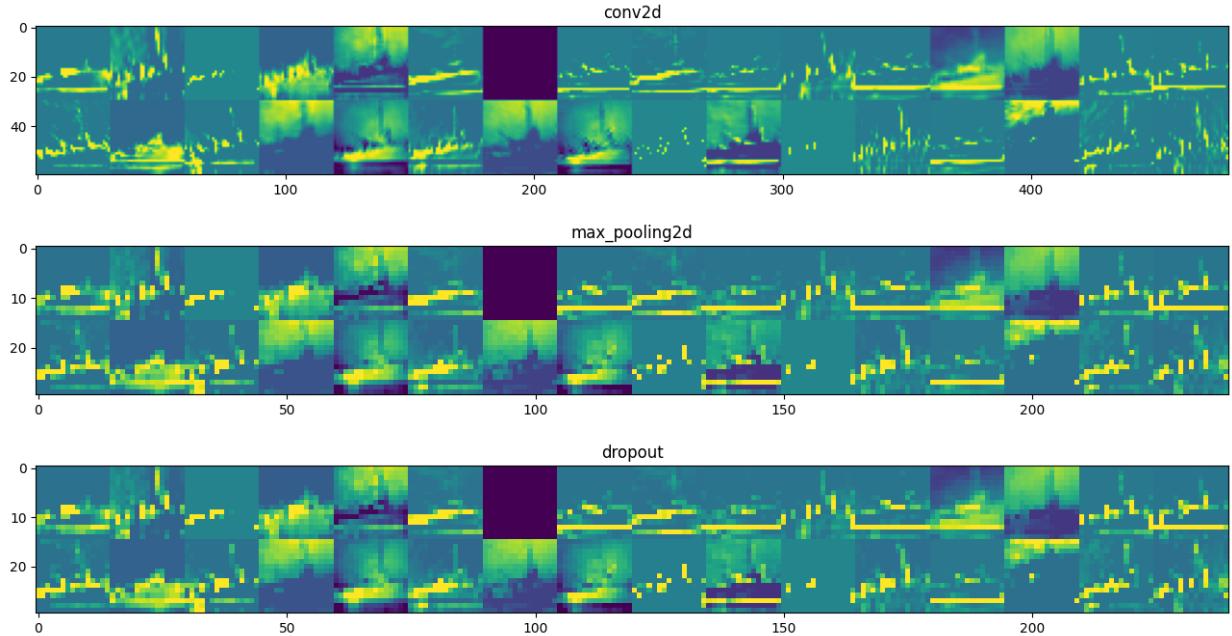
plt.show();
```



1/1 [=====] - 0s 89ms/step

<ipython-input-302-f532589aadeb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[133]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat']
```

```

        , 'deer'
        , 'dog'
        , 'frog'
        , 'horse'
        , 'ship'
        , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

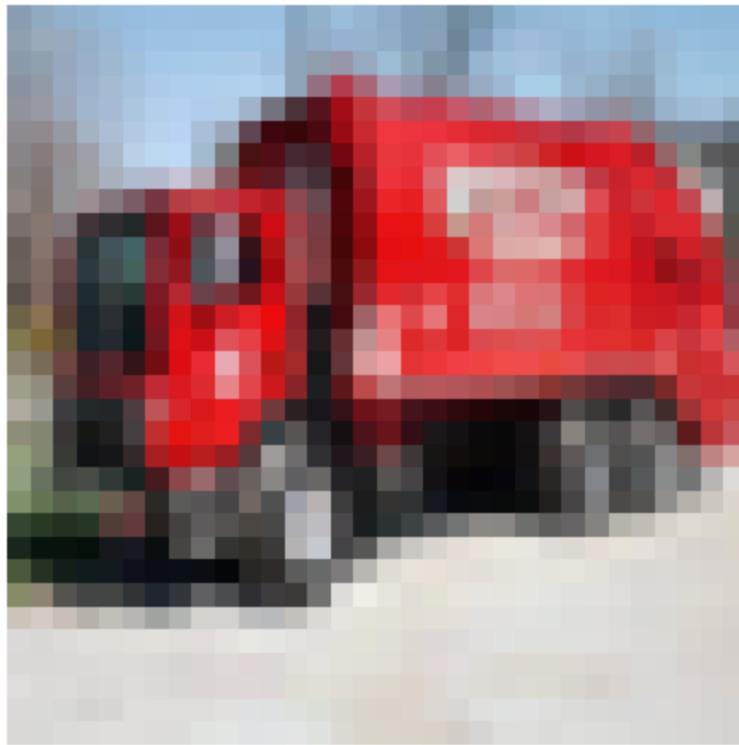
    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable

```

```
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

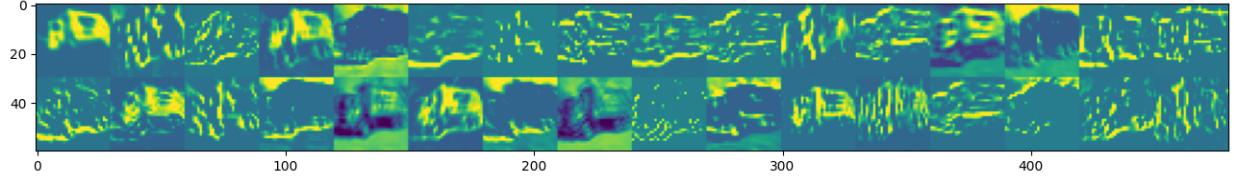
# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

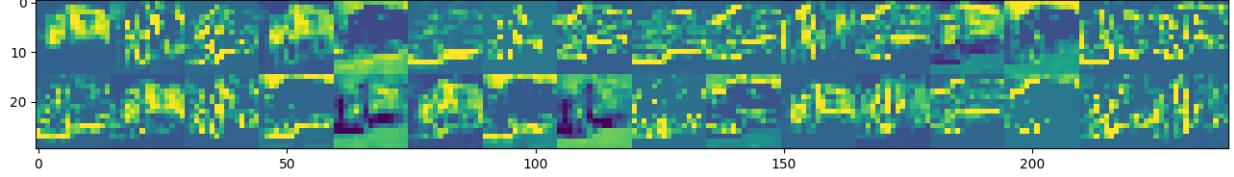


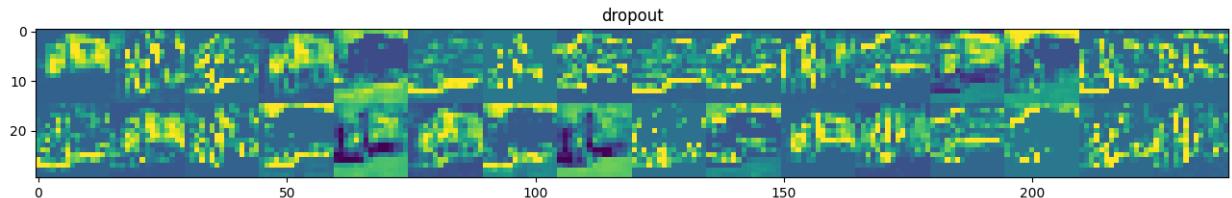
1/1 [=====] - 0s 73ms/step

conv2d



max_pooling2d





In []:

10) Model 9 - CNN with 3 Max Pooling / Hidden Layers and Early Stopping, Batch Normalization, Dropout, and L1 Regularization

10.1 Build The Model

We use a Sequential class defined in Keras to create our model.

```
In [ ]: k.clear_session()
model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu),
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Flatten(),
    Dense(units=256,activation=tf.nn.softmax,kernel_regularizer=tf.keras.regularizers.L1
    BatchNormalization(),
    Dropout(0.3),
```

```
Dense(units=10, activation=tf.nn.softmax)  
])
```

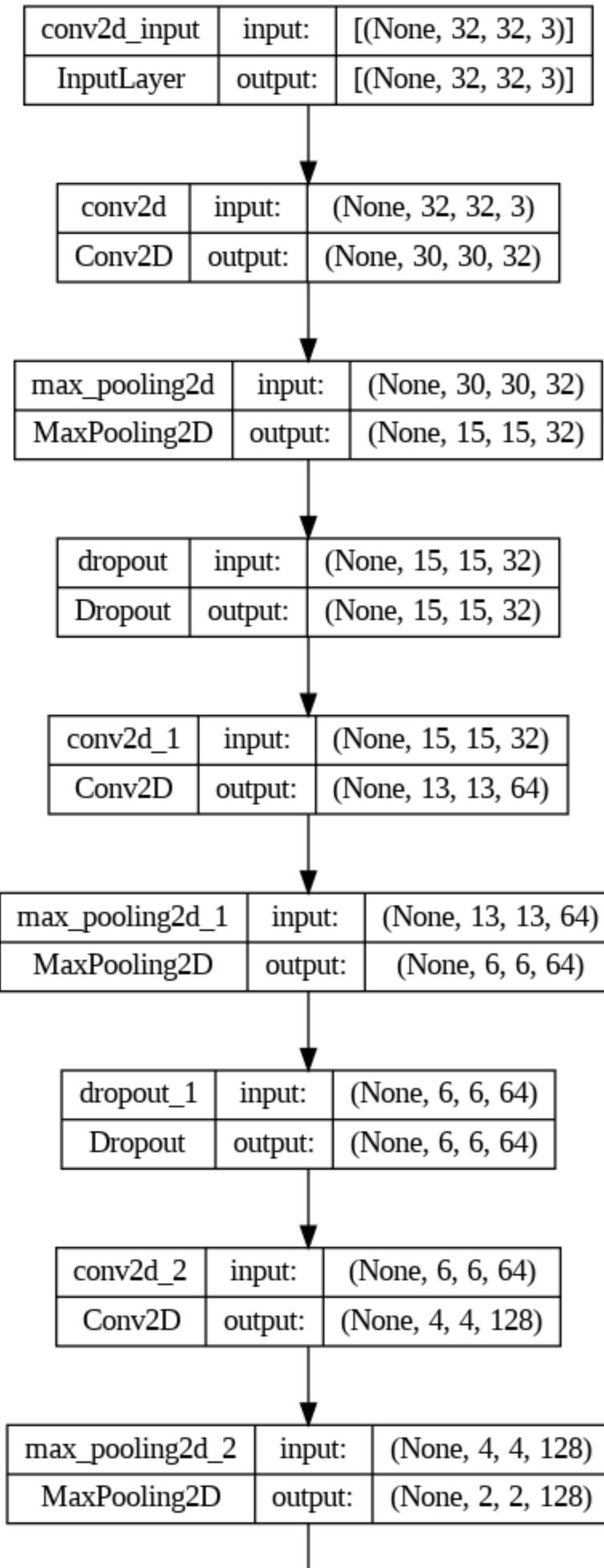
In []: `model.summary()`

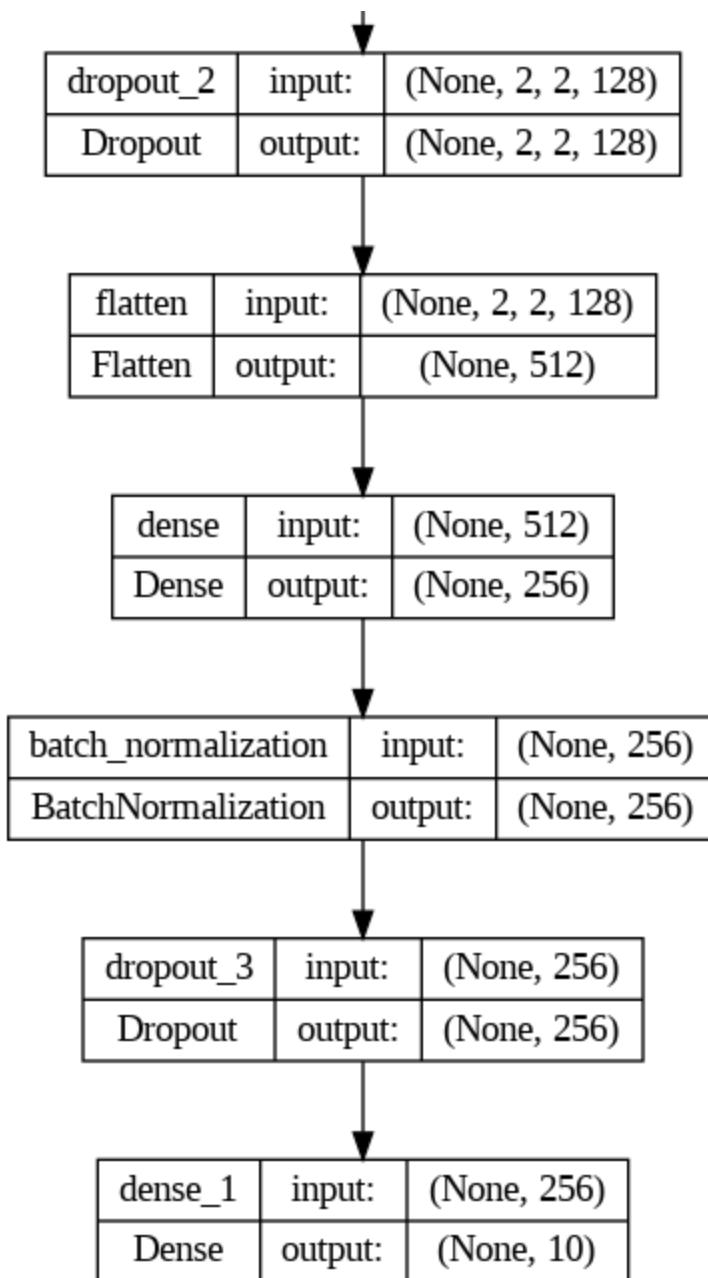
Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
dropout (Dropout)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
dropout_1 (Dropout)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
dropout_2 (Dropout)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 256)	131328
batch_normalization (Batch Normalization)	(None, 256)	1024
dropout_3 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2570
<hr/>		
Total params: 228170 (891.29 KB)		
Trainable params: 227658 (889.29 KB)		
Non-trainable params: 512 (2.00 KB)		

In []: `tf.keras.utils.plot_model(model, "CIFAR10.png", show_shapes=True)`

Out[]:





Let's now compile and train the model.

tf.keras.losses.SparseCategoricalCrossentropy

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Module: tf.keras.callbacks

tf.keras.callbacks.EarlyStopping

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

tf.keras.callbacks.ModelCheckpointhttps://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

```
In [ ]: model.compile(optimizer='adam',
                      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
                      metrics=['accuracy'])
```

```
In [ ]: history = model.fit(x_train_norm
                           ,y_train_split
                           ,epochs=40
                           ,batch_size=500
                           ,validation_data=(x_valid_norm, y_valid_split)
                           ,callbacks=[
                           tf.keras.callbacks.ModelCheckpoint("Model_9", save_best_only=True,
                           ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=10)
                           ])
```

```
Epoch 1/40
90/90 [=====] - 42s 453ms/step - loss: 4.1477 - accuracy: 0.
2082 - val_loss: 2.4153 - val_accuracy: 0.1364
Epoch 2/40
90/90 [=====] - 41s 454ms/step - loss: 1.9237 - accuracy: 0.
3087 - val_loss: 2.2770 - val_accuracy: 0.1492
Epoch 3/40
90/90 [=====] - 40s 450ms/step - loss: 1.7714 - accuracy: 0.
3720 - val_loss: 2.2048 - val_accuracy: 0.1898
Epoch 4/40
90/90 [=====] - 40s 447ms/step - loss: 1.7014 - accuracy: 0.
4017 - val_loss: 2.1079 - val_accuracy: 0.2710
Epoch 5/40
90/90 [=====] - 41s 452ms/step - loss: 1.6431 - accuracy: 0.
4272 - val_loss: 1.8689 - val_accuracy: 0.4198
Epoch 6/40
90/90 [=====] - 40s 449ms/step - loss: 1.5903 - accuracy: 0.
4491 - val_loss: 1.6761 - val_accuracy: 0.4816
Epoch 7/40
90/90 [=====] - 41s 456ms/step - loss: 1.5437 - accuracy: 0.
4733 - val_loss: 1.5679 - val_accuracy: 0.4990
Epoch 8/40
90/90 [=====] - 41s 456ms/step - loss: 1.4980 - accuracy: 0.
4916 - val_loss: 1.5047 - val_accuracy: 0.4790
Epoch 9/40
90/90 [=====] - 40s 449ms/step - loss: 1.4733 - accuracy: 0.
5063 - val_loss: 1.4079 - val_accuracy: 0.5130
Epoch 10/40
90/90 [=====] - 40s 446ms/step - loss: 1.4399 - accuracy: 0.
5161 - val_loss: 1.5102 - val_accuracy: 0.4892
Epoch 11/40
90/90 [=====] - 40s 445ms/step - loss: 1.4203 - accuracy: 0.
5255 - val_loss: 1.3774 - val_accuracy: 0.5372
Epoch 12/40
90/90 [=====] - 39s 437ms/step - loss: 1.3946 - accuracy: 0.
5336 - val_loss: 1.4504 - val_accuracy: 0.5154
Epoch 13/40
90/90 [=====] - 40s 450ms/step - loss: 1.3787 - accuracy: 0.
5422 - val_loss: 1.3574 - val_accuracy: 0.5434
Epoch 14/40
90/90 [=====] - 40s 446ms/step - loss: 1.3559 - accuracy: 0.
5492 - val_loss: 1.3168 - val_accuracy: 0.5648
Epoch 15/40
90/90 [=====] - 44s 492ms/step - loss: 1.3343 - accuracy: 0.
5625 - val_loss: 1.2732 - val_accuracy: 0.5686
Epoch 16/40
90/90 [=====] - 40s 441ms/step - loss: 1.3172 - accuracy: 0.
5676 - val_loss: 1.3239 - val_accuracy: 0.5674
Epoch 17/40
90/90 [=====] - 40s 447ms/step - loss: 1.3143 - accuracy: 0.
5714 - val_loss: 1.2827 - val_accuracy: 0.5708
Epoch 18/40
90/90 [=====] - 40s 449ms/step - loss: 1.2934 - accuracy: 0.
5769 - val_loss: 1.2292 - val_accuracy: 0.6002
Epoch 19/40
90/90 [=====] - 41s 454ms/step - loss: 1.2749 - accuracy: 0.
5848 - val_loss: 1.2083 - val_accuracy: 0.6032
Epoch 20/40
90/90 [=====] - 41s 452ms/step - loss: 1.2628 - accuracy: 0.
5863 - val_loss: 1.1761 - val_accuracy: 0.6184
```

```

Epoch 21/40
90/90 [=====] - 40s 449ms/step - loss: 1.2546 - accuracy: 0.
5928 - val_loss: 1.1436 - val_accuracy: 0.6334
Epoch 22/40
90/90 [=====] - 39s 437ms/step - loss: 1.2445 - accuracy: 0.
5964 - val_loss: 1.2794 - val_accuracy: 0.5772
Epoch 23/40
90/90 [=====] - 41s 453ms/step - loss: 1.2335 - accuracy: 0.
6014 - val_loss: 1.1213 - val_accuracy: 0.6404
Epoch 24/40
90/90 [=====] - 39s 439ms/step - loss: 1.2192 - accuracy: 0.
6046 - val_loss: 1.1997 - val_accuracy: 0.6120
Epoch 25/40
90/90 [=====] - 39s 433ms/step - loss: 1.2162 - accuracy: 0.
6075 - val_loss: 1.2778 - val_accuracy: 0.5740
Epoch 26/40
90/90 [=====] - 39s 434ms/step - loss: 1.2014 - accuracy: 0.
6148 - val_loss: 1.2034 - val_accuracy: 0.5944

```

10.2) Evaluate Model Performance

```

In [ ]: model = tf.keras.models.load_model("Model_9")
print(f"Training accuracy: {model.evaluate(x_train_norm, y_train_split)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_norm, y_valid_split)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_norm, y_test)[1]:.3f}")

1407/1407 [=====] - 20s 14ms/step - loss: 1.0941 - accuracy: 0.6556
Training accuracy: 0.656
157/157 [=====] - 3s 17ms/step - loss: 1.1213 - accuracy: 0.6404
Validation accuracy: 0.640
313/313 [=====] - 4s 13ms/step - loss: 1.1453 - accuracy: 0.6374
Test accuracy: 0.637

In [ ]: preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)

313/313 [=====] - 3s 9ms/step
shape of preds: (10000, 10)

In [ ]: history_dict = history.history
history_dict.keys()

Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])

In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

```

Out[]:

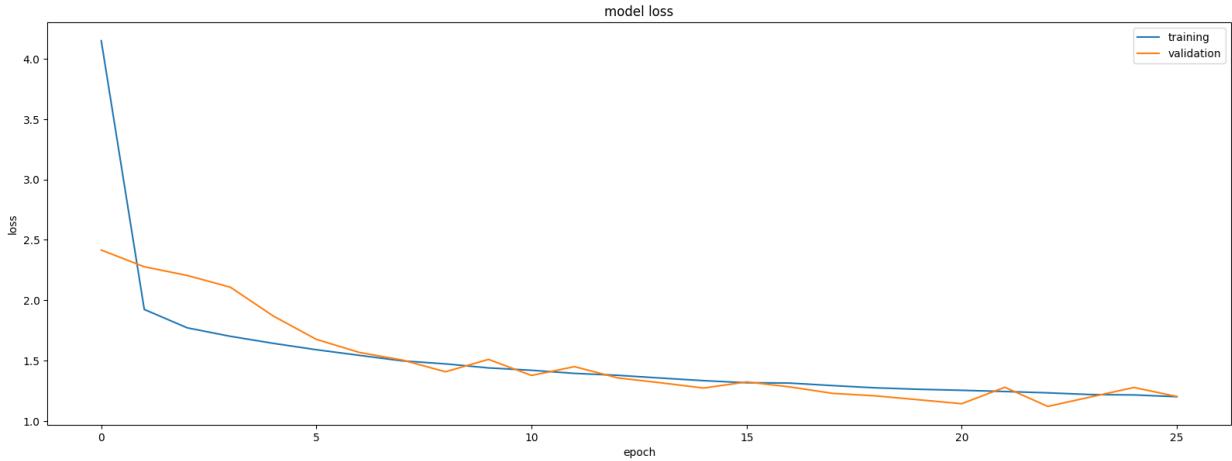
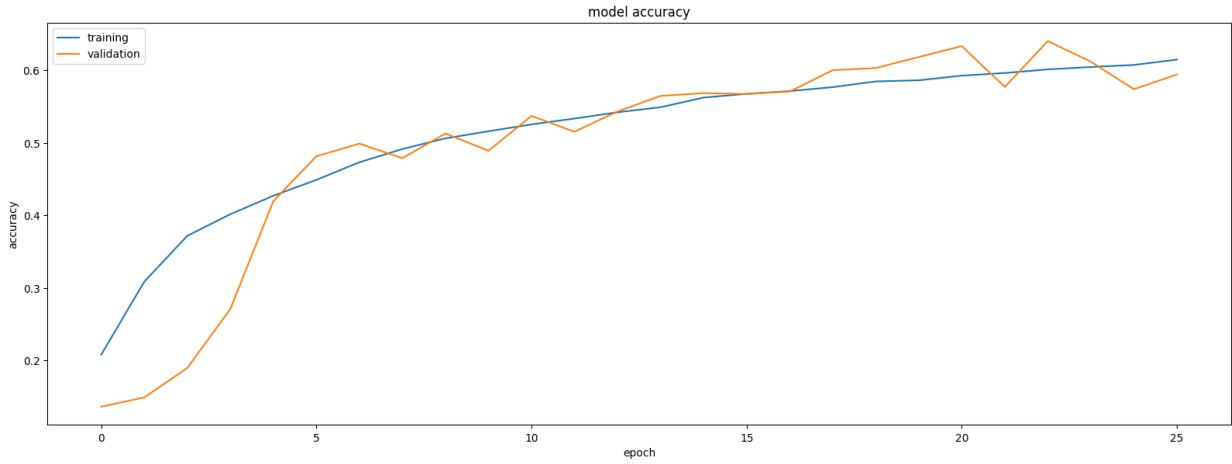
	loss	accuracy	val_loss	val_accuracy
21	1.244	0.596	1.279	0.577
22	1.234	0.601	1.121	0.640
23	1.219	0.605	1.200	0.612
24	1.216	0.607	1.278	0.574
25	1.201	0.615	1.203	0.594

In []:

```
plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```

<ipython-input-8-353fbbe40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.

```
ax = plt.subplot(subplot)
```



Let's examine the precision, recall, F1 score, and confusion matrix.

In []:

```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)
```

```
313/313 [=====] - 3s 11ms/step
```

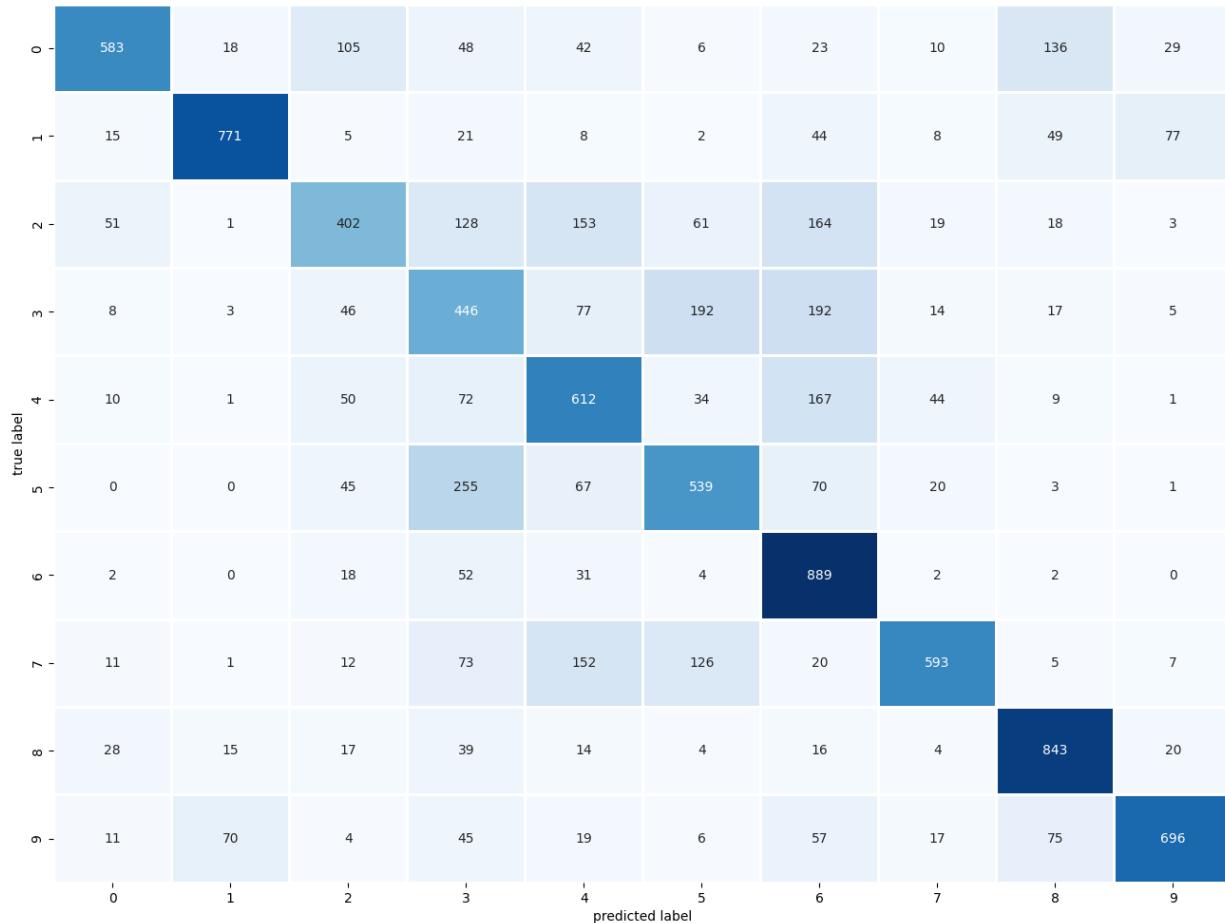
```
In [ ]: print_validation_report(y_test, pred1)
```

		precision	recall	f1-score	support
	0	0.81	0.58	0.68	1000
	1	0.88	0.77	0.82	1000
	2	0.57	0.40	0.47	1000
	3	0.38	0.45	0.41	1000
	4	0.52	0.61	0.56	1000
	5	0.55	0.54	0.55	1000
	6	0.54	0.89	0.67	1000
	7	0.81	0.59	0.69	1000
	8	0.73	0.84	0.78	1000
	9	0.83	0.70	0.76	1000
				accuracy	0.64
				macro avg	0.64
				weighted avg	0.64

Accuracy Score: 0.6374

Root Mean Square Error: 2.304582391670994

```
In [ ]: plot_confusion_matrix(y_test,pred1)
```



Load HDF5 Model Format

tf.keras.models.load_modelhttps://www.tensorflow.org/api_docs/python/tf/keras/models/load_model

```
In [ ]: model = tf.keras.models.load_model('Model_9')
preds = model.predict(x_test_norm)
preds.shape

313/313 [=====] - 4s 12ms/step
(10000, 10)

Out[ ]:
```

Let's examine the predictions for the testing dataset.

```
In [ ]: cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)

df = pd.DataFrame(preds[0:20], columns = ['airplane'
                                             , 'automobile'
                                             , 'bird'
                                             , 'cat'
                                             , 'deer'
                                             , 'dog'
                                             , 'frog'
                                             , 'horse'
                                             , 'ship'
                                             , 'truck'])

df.style.format("{:.2%}").background_gradient(cmap=cm)
```

Out[]:

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0	2.61%	1.39%	4.94%	47.50%	0.69%	15.52%	6.09%	0.77%	19.52%	0.98%
1	4.66%	3.86%	0.01%	0.01%	0.00%	0.00%	0.00%	0.00%	91.06%	0.39%
2	7.06%	7.30%	1.06%	1.42%	0.52%	0.28%	0.52%	0.21%	77.90%	3.73%
3	52.88%	8.66%	2.23%	0.39%	0.79%	0.14%	0.24%	0.18%	32.98%	1.51%
4	0.03%	0.04%	3.36%	2.48%	11.92%	0.81%	81.23%	0.09%	0.03%	0.01%
5	0.04%	0.06%	2.69%	5.61%	3.28%	2.00%	86.02%	0.22%	0.05%	0.04%
6	4.56%	11.40%	5.72%	30.38%	1.78%	24.86%	11.05%	3.65%	1.36%	5.25%
7	0.21%	0.06%	13.06%	3.77%	27.84%	1.43%	53.23%	0.26%	0.10%	0.03%
8	0.11%	0.07%	6.86%	41.29%	5.97%	31.27%	12.57%	1.69%	0.10%	0.06%
9	2.03%	68.32%	0.24%	0.13%	0.43%	0.06%	0.49%	0.35%	1.11%	26.83%
10	25.96%	0.23%	25.37%	5.68%	29.62%	3.68%	0.45%	1.53%	7.26%	0.22%
11	0.15%	0.45%	0.00%	0.07%	0.00%	0.01%	0.01%	0.03%	2.17%	97.11%
12	0.19%	0.16%	7.85%	33.25%	10.44%	28.85%	15.90%	3.11%	0.14%	0.11%
13	0.02%	0.02%	0.31%	0.68%	2.40%	5.17%	0.00%	91.31%	0.00%	0.08%
14	1.13%	2.16%	0.06%	0.15%	0.12%	0.04%	0.06%	0.32%	5.01%	90.96%
15	4.65%	1.27%	10.81%	11.10%	10.67%	2.37%	39.86%	0.41%	18.17%	0.68%
16	0.02%	0.01%	0.31%	24.75%	0.02%	74.33%	0.03%	0.50%	0.02%	0.01%
17	1.70%	0.98%	8.65%	25.02%	16.21%	20.20%	3.62%	19.48%	0.92%	3.22%
18	1.76%	1.89%	0.01%	0.06%	0.01%	0.01%	0.02%	0.01%	86.27%	9.96%
19	0.01%	0.05%	1.75%	1.60%	5.34%	0.60%	90.53%	0.10%	0.01%	0.01%

In []:

```
(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[98]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
,'automobile'
,'bird'
,'cat'
,'deer'
,'dog'
,'frog'
,'horse'
,'ship'
,'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))

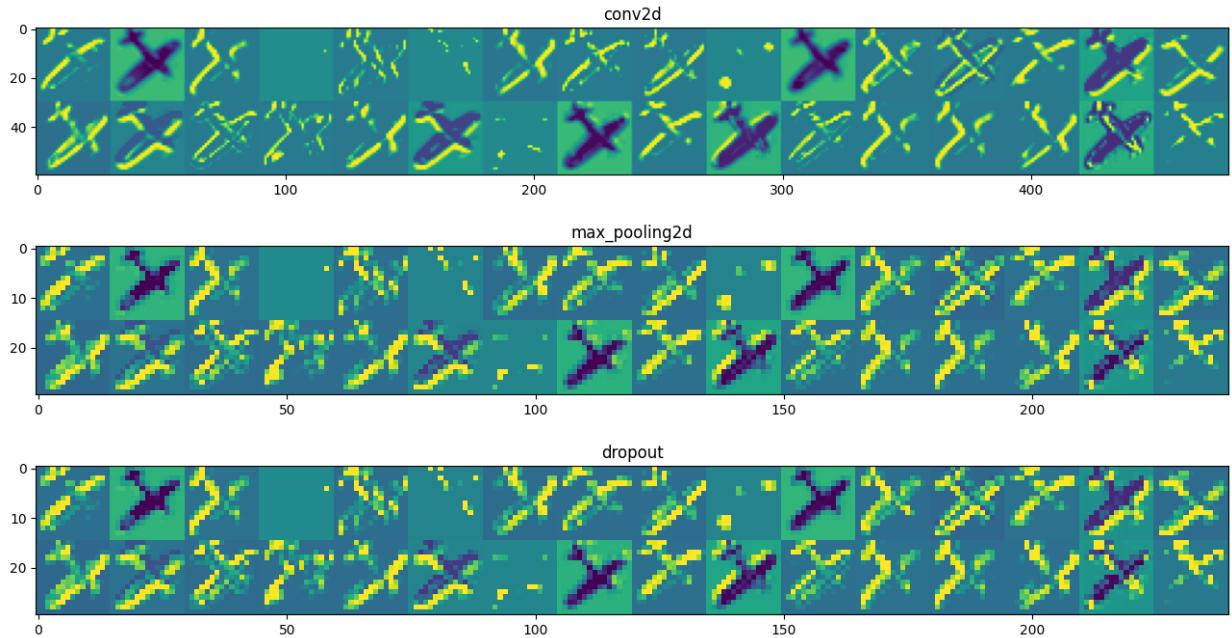
```

```
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```



1/1 [=====] - 0s 163ms/step



In []: `(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()`

```
img = test_images[122]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile'
```

```
, 'bird'
, 'cat'
, 'deer'
, 'dog'
, 'frog'
, 'horse'
, 'ship'
, 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

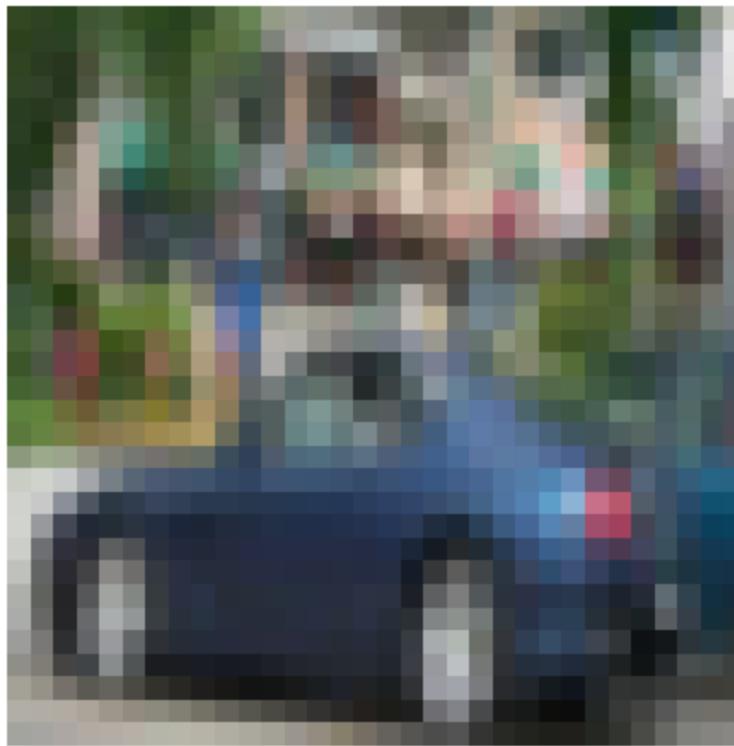
    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
```

```
col * images_per_row + row]
# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

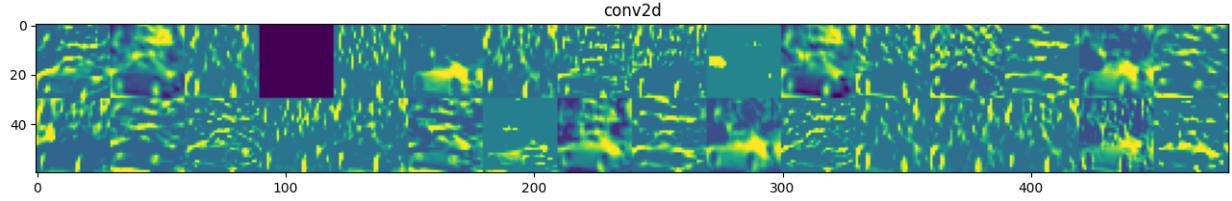
plt.show();
```

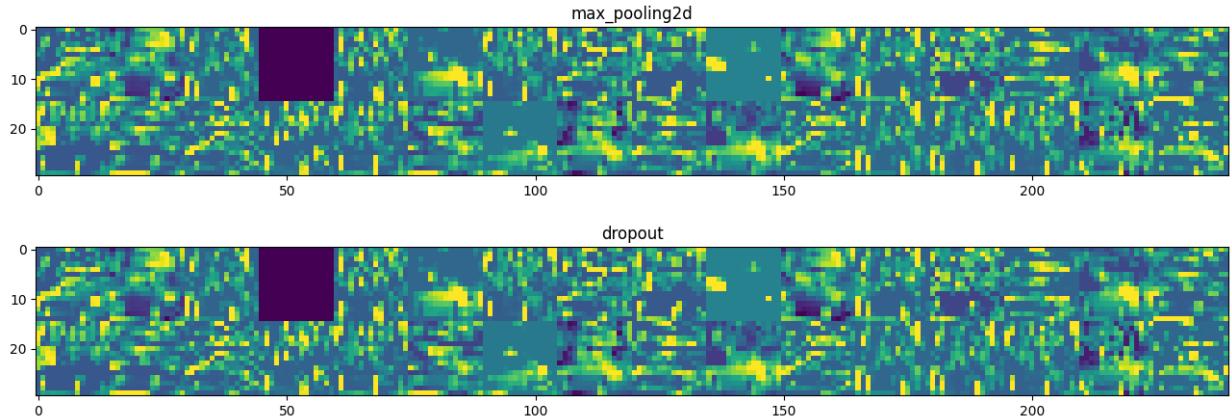


1/1 [=====] - 0s 105ms/step

<ipython-input-326-e0d043c5b9b7>:72: RuntimeWarning: invalid value encountered in divide

```
channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[75]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
```

```
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

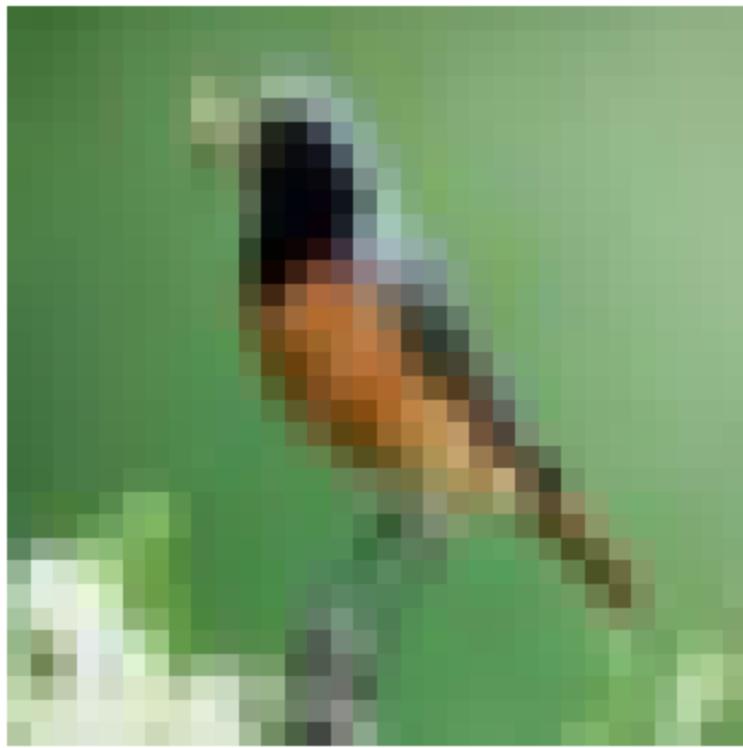
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

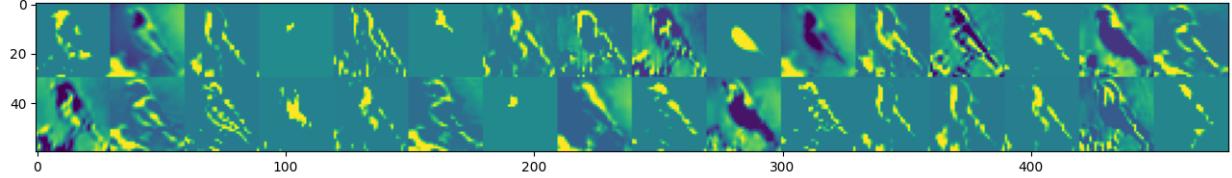
    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

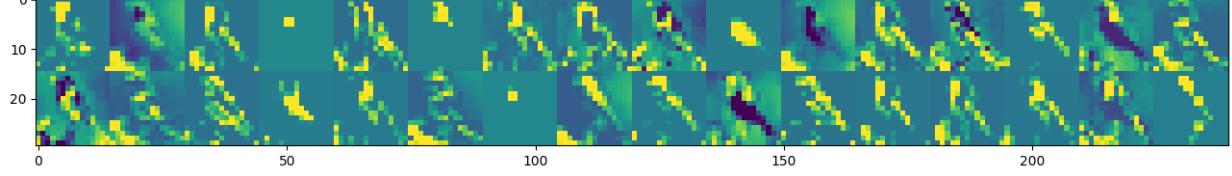


1/1 [=====] - 0s 69ms/step

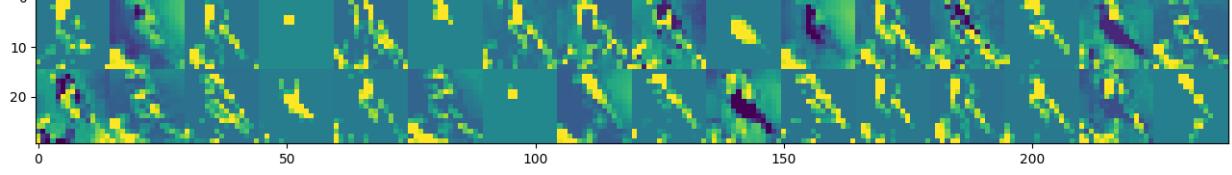
conv2d



max_pooling2d



dropout



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[184]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
,'automobile'
,'bird'
,'cat'
,'deer'
,'dog'
,'frog'
```

```

    , 'horse'
    , 'ship'
    , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

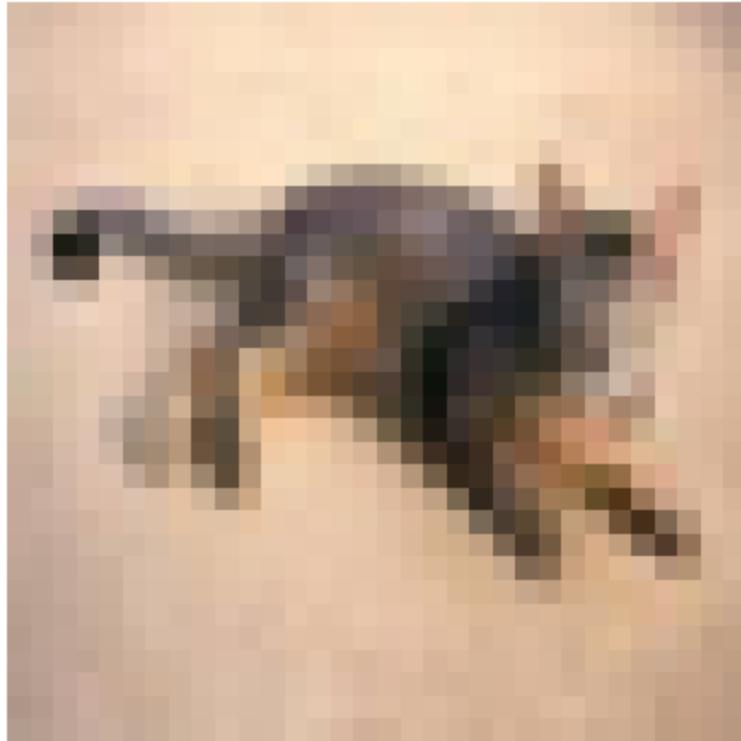
    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64

```

```
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                    scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

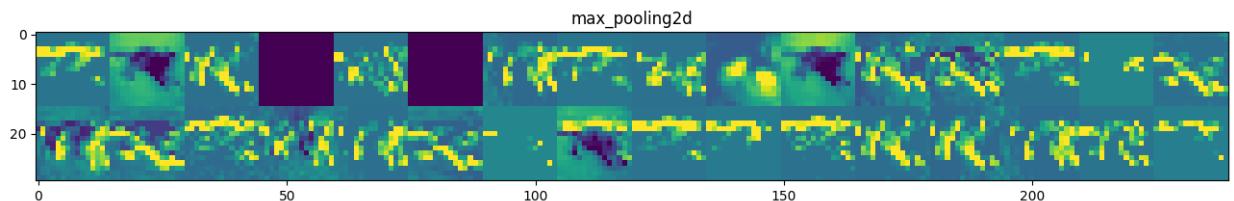
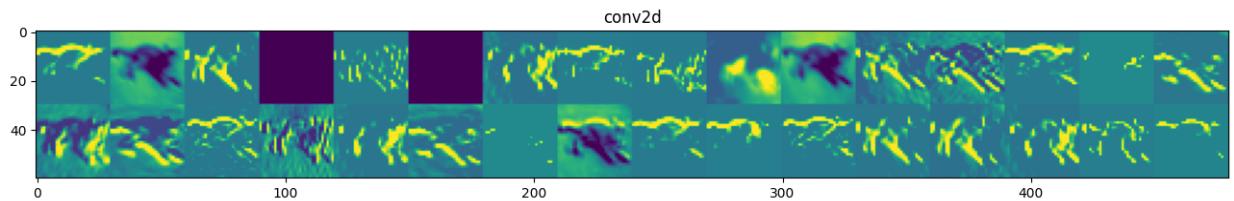
plt.show();
```

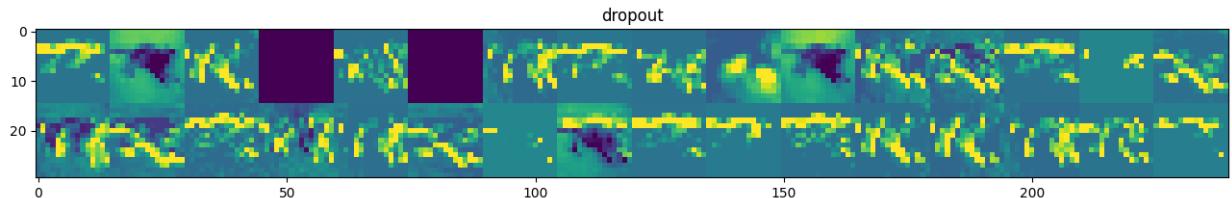


1/1 [=====] - 0s 84ms/step

<ipython-input-328-3bac8bdd9965>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[159]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
```

```
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

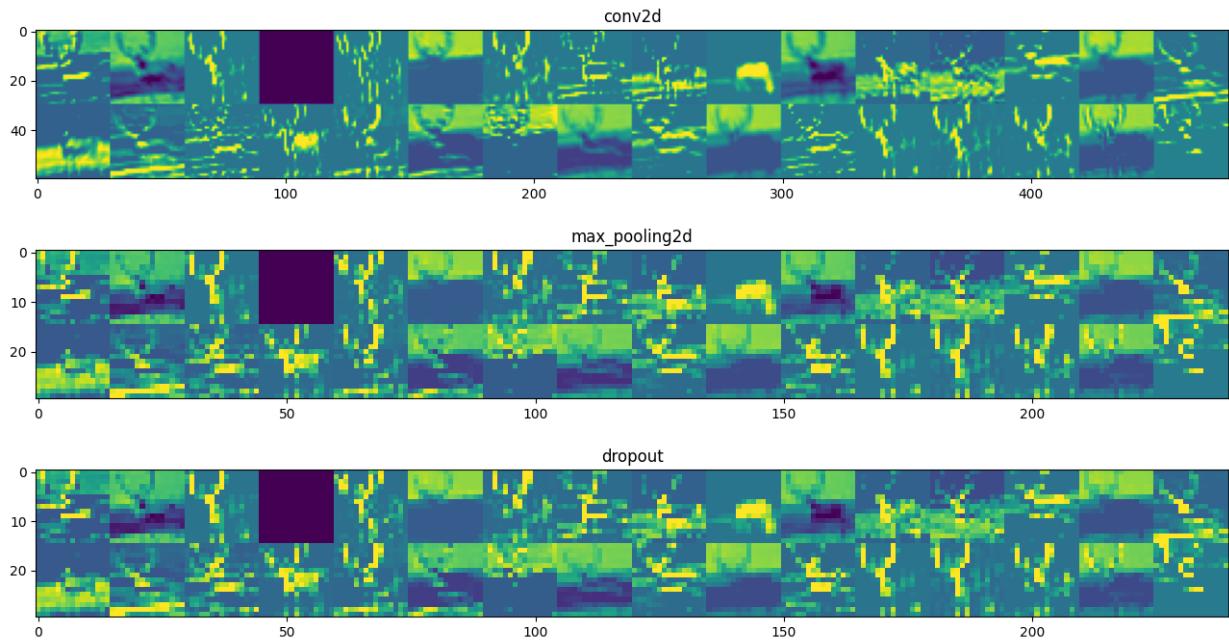
plt.show();
```



1/1 [=====] - 0s 89ms/step

<ipython-input-329-a52112e96c59>:72: RuntimeWarning: invalid value encountered in divide

channel_image /= channel_image.std()

In []: `(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()`

```
img = test_images[24]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

```
plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

Extracts the outputs of the top 8 layers:

```
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

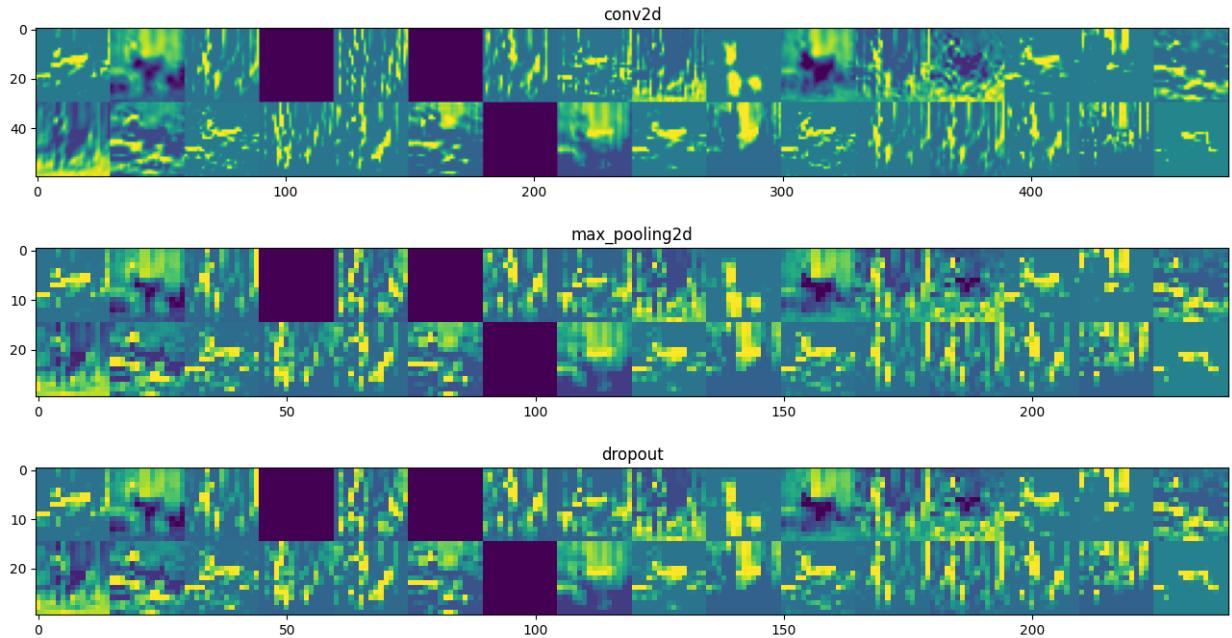
plt.show();
```



```
1/1 [=====] - 0s 93ms/step
```

```
<ipython-input-330-edee34e9d17b>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[152]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat']
```

```

        , 'deer'
        , 'dog'
        , 'frog'
        , 'horse'
        , 'ship'
        , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable

```

```
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

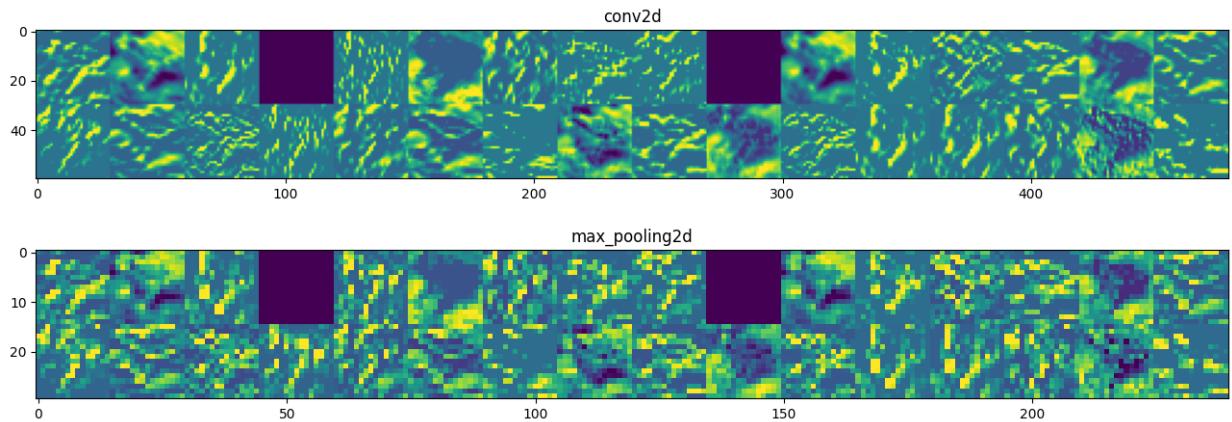
plt.show();
```

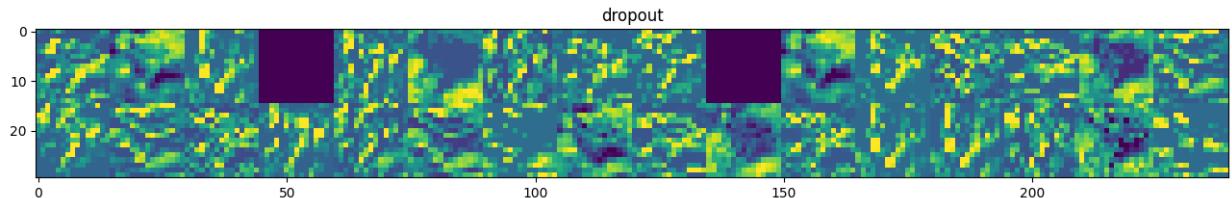


1/1 [=====] - 0s 84ms/step

<ipython-input-331-0ef3f9c61bfb>:72: RuntimeWarning: invalid value encountered in divide

```
    channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[2004]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
```

```
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

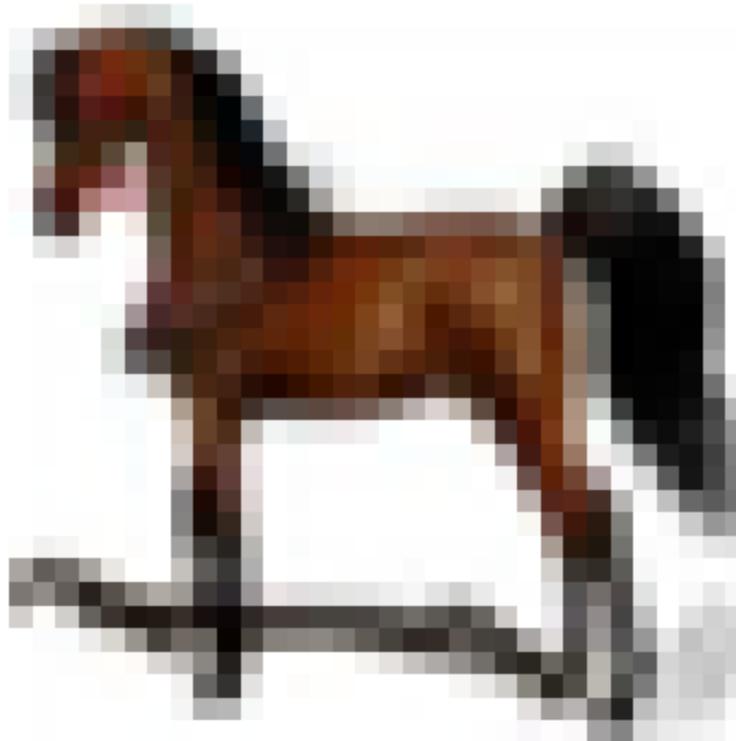
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

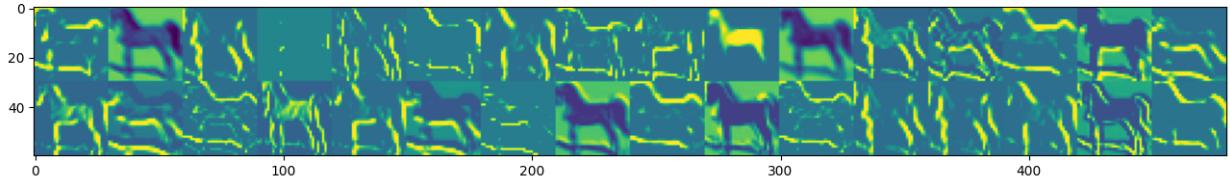
    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

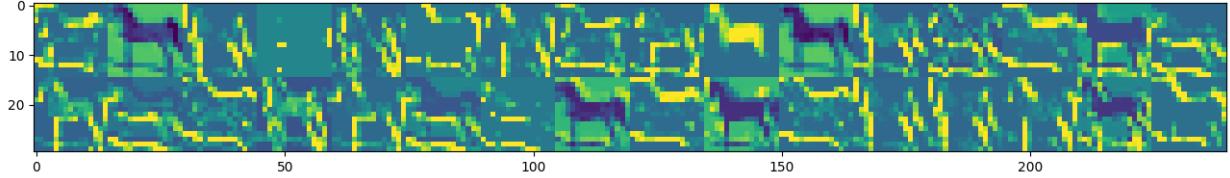


1/1 [=====] - 0s 86ms/step

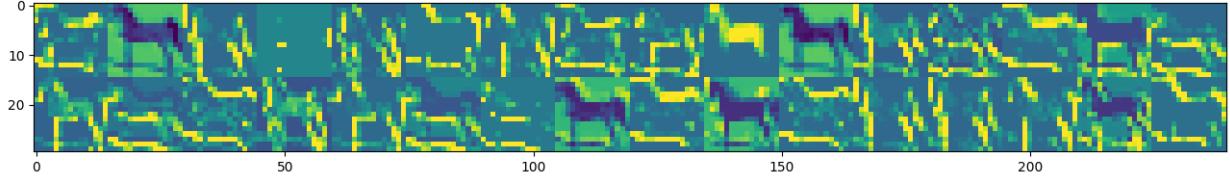
conv2d



max_pooling2d



dropout

In []:

```
(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[185]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

Extracts the outputs of the top 8 Layers:

```
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

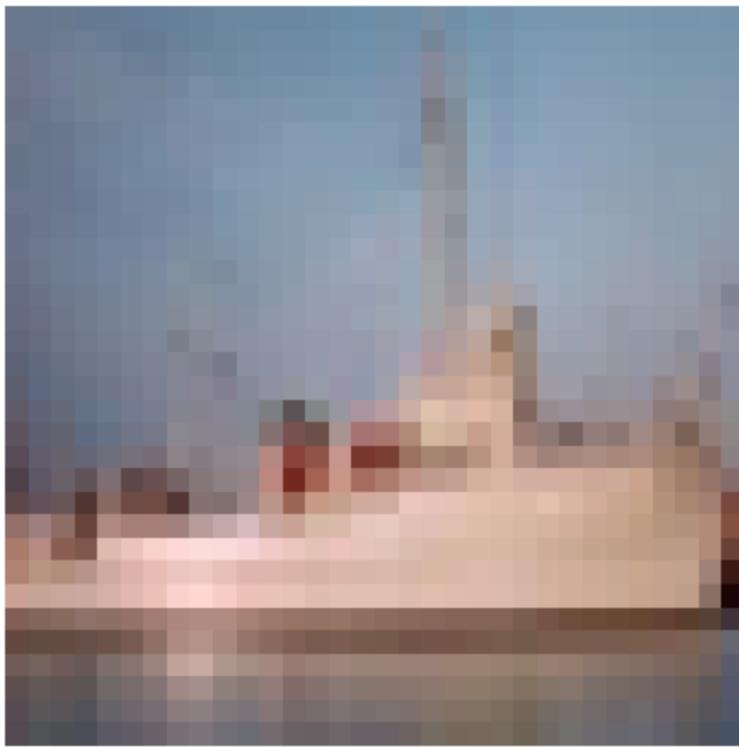
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

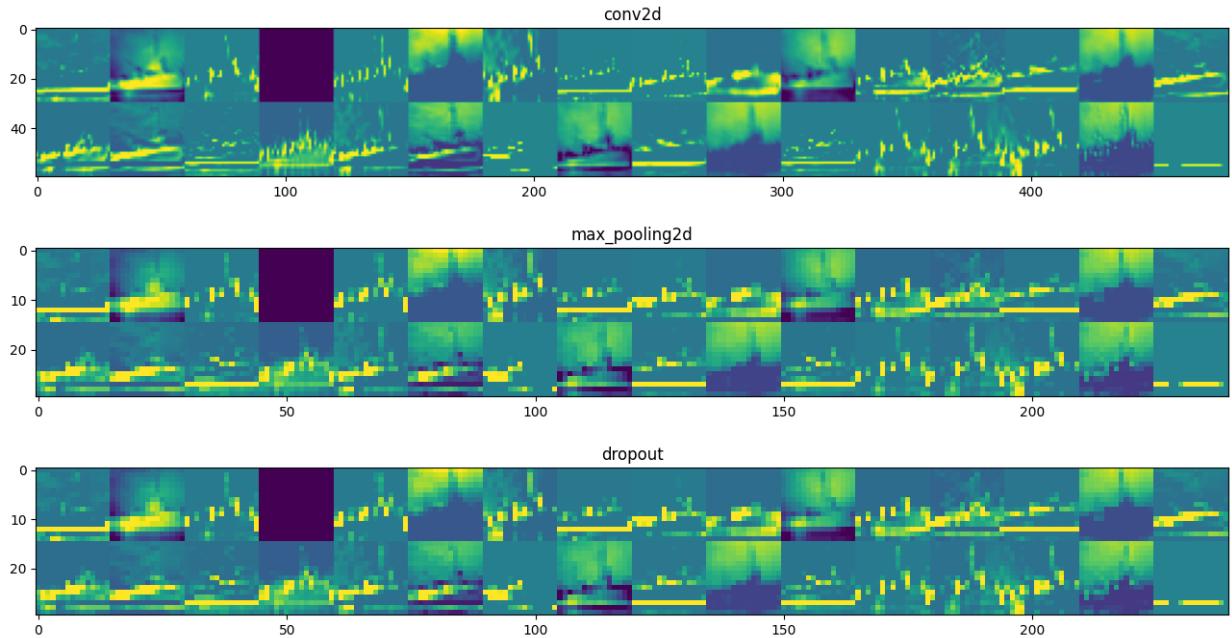
plt.show();
```



```
1/1 [=====] - 0s 180ms/step
```

```
<ipython-input-333-f532589aadeb>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[133]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat']
```

```

        , 'deer'
        , 'dog'
        , 'frog'
        , 'horse'
        , 'ship'
        , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

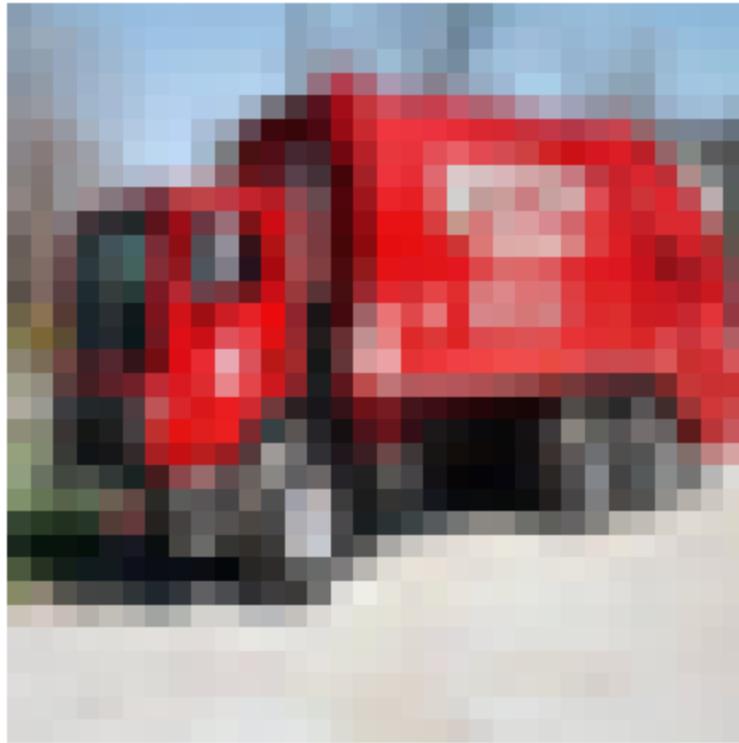
    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable

```

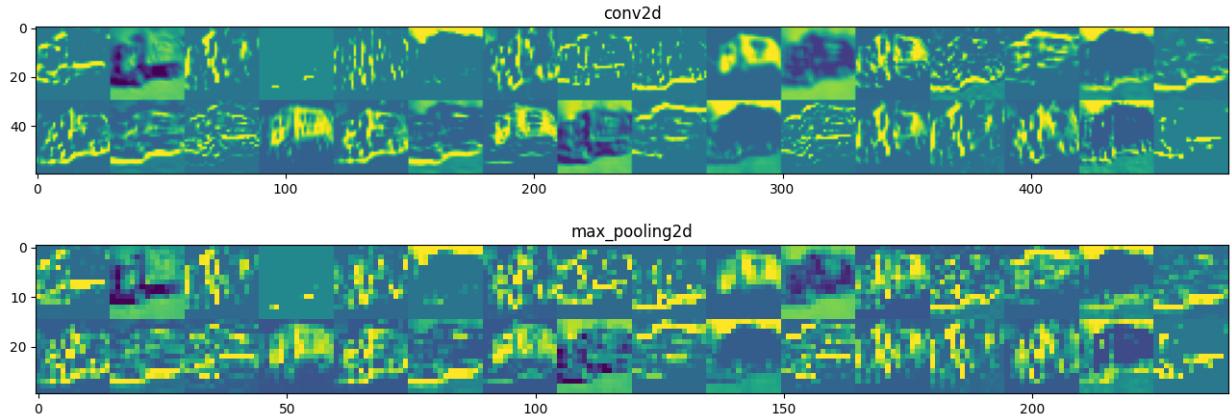
```
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

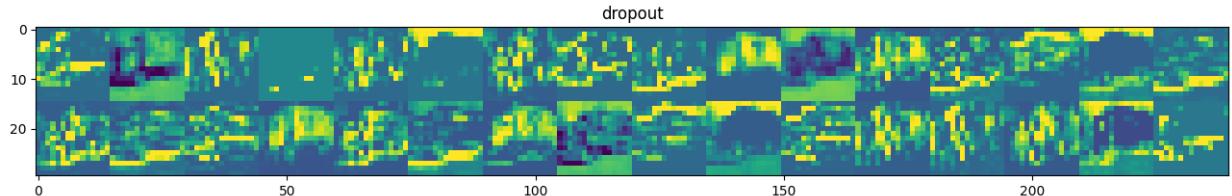
# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                    scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```



1/1 [=====] - 0s 82ms/step





In []:

11) Model 10 - CNN with 3 Max Pooling / Hidden Layers and Regularizatoin via Early Stopping, Batch Normalization, Dropout, and More Simple Architecture

11.1) Build The Model

We use a Sequential class defined in Keras to create our model.

```
In [ ]: k.clear_session()
model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Conv2D(filters=64, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu, input_s
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Conv2D(filters=128, kernel_size=(3, 3), strides=(1, 1), activation=tf.nn.relu),
    MaxPool2D((2, 2),strides=2),
    Dropout(0.3),
    Flatten(),
    Dense(units=128,activation=tf.nn.relu),
    BatchNormalization(),
    Dropout(0.3),
```

```
Dense(units=10, activation=tf.nn.softmax)  
])
```

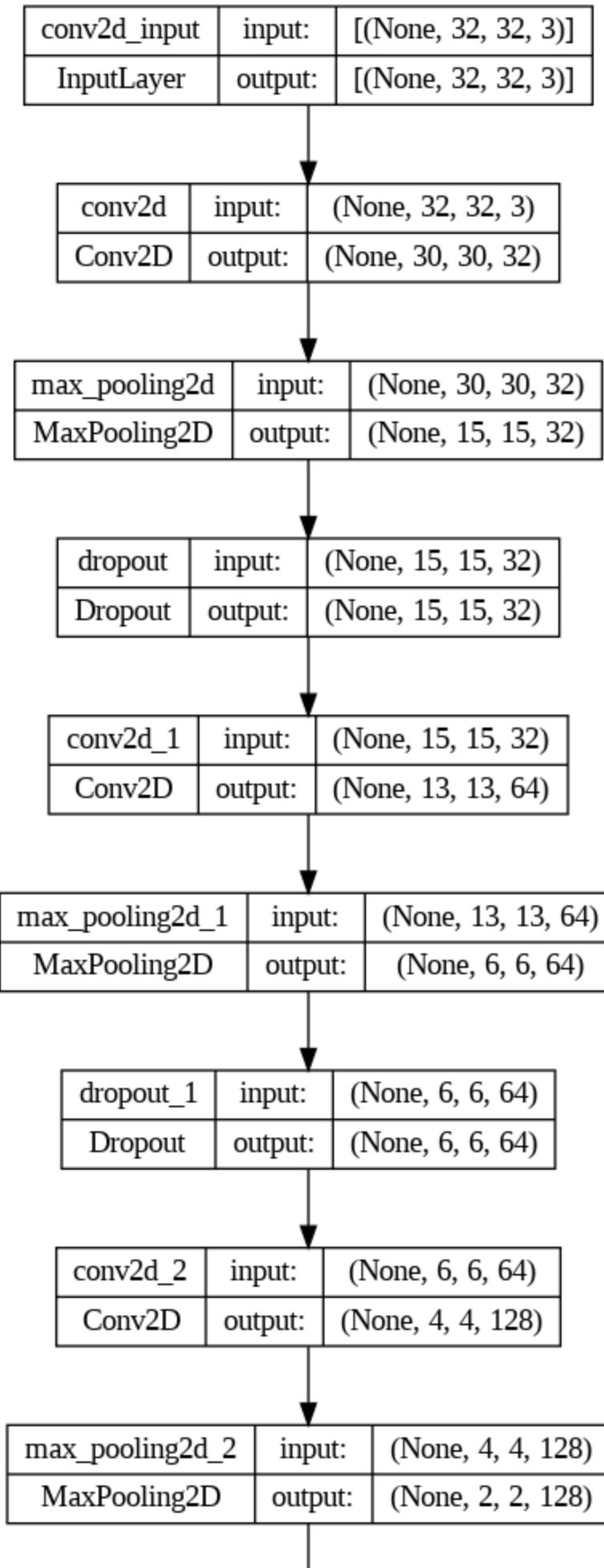
In []: `model.summary()`

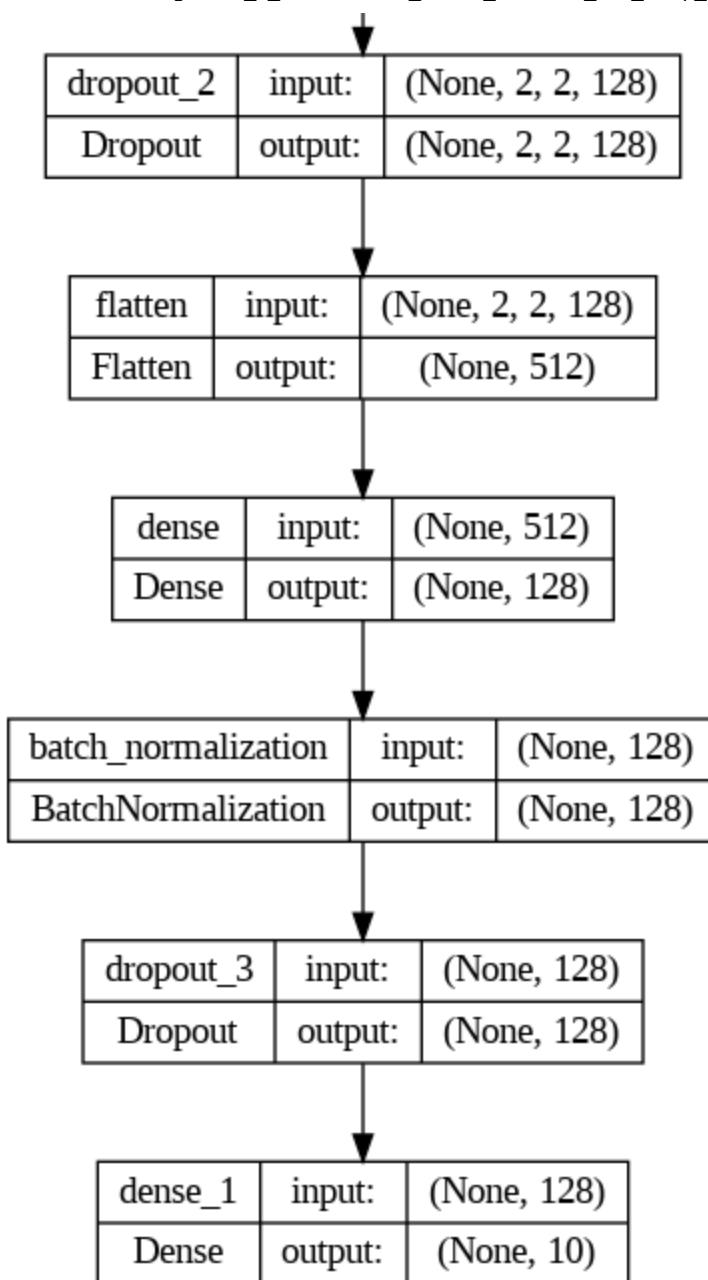
Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
dropout (Dropout)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
dropout_1 (Dropout)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
dropout_2 (Dropout)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 128)	65664
batch_normalization (Batch Normalization)	(None, 128)	512
dropout_3 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1290
<hr/>		
Total params: 160714 (627.79 KB)		
Trainable params: 160458 (626.79 KB)		
Non-trainable params: 256 (1.00 KB)		

In []: `tf.keras.utils.plot_model(model, "CIFAR10.png", show_shapes=True)`

Out[]:





Let's now compile and train the model.

tf.keras.losses.SparseCategoricalCrossentropy

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Module: tf.keras.callbacks

tf.keras.callbacks.EarlyStopping

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

tf.keras.callbacks.ModelCheckpointhttps://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

```
In [ ]: model.compile(optimizer='adam',
                      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
                      metrics=['accuracy'])
```

```
In [ ]: history = model.fit(x_train_norm
                           ,y_train_split
                           ,epochs=40
                           ,batch_size=500
                           ,validation_data=(x_valid_norm, y_valid_split)
                           ,callbacks=[
                           tf.keras.callbacks.ModelCheckpoint("Model_10", save_best_only=True,
                           ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience
                           ]
                           )
```

```
Epoch 1/40
90/90 [=====] - 74s 783ms/step - loss: 2.0939 - accuracy: 0.
2589 - val_loss: 2.2148 - val_accuracy: 0.1952
Epoch 2/40
90/90 [=====] - 57s 639ms/step - loss: 1.5764 - accuracy: 0.
4181 - val_loss: 2.1117 - val_accuracy: 0.2486
Epoch 3/40
90/90 [=====] - 52s 574ms/step - loss: 1.4170 - accuracy: 0.
4854 - val_loss: 1.9595 - val_accuracy: 0.3266
Epoch 4/40
90/90 [=====] - 41s 457ms/step - loss: 1.3035 - accuracy: 0.
5314 - val_loss: 1.6939 - val_accuracy: 0.5184
Epoch 5/40
90/90 [=====] - 44s 484ms/step - loss: 1.2350 - accuracy: 0.
5585 - val_loss: 1.4017 - val_accuracy: 0.5952
Epoch 6/40
90/90 [=====] - 43s 475ms/step - loss: 1.1791 - accuracy: 0.
5800 - val_loss: 1.2189 - val_accuracy: 0.5842
Epoch 7/40
90/90 [=====] - 42s 469ms/step - loss: 1.1309 - accuracy: 0.
5992 - val_loss: 1.1676 - val_accuracy: 0.5884
Epoch 8/40
90/90 [=====] - 47s 520ms/step - loss: 1.0969 - accuracy: 0.
6140 - val_loss: 1.1083 - val_accuracy: 0.6024
Epoch 9/40
90/90 [=====] - 41s 455ms/step - loss: 1.0592 - accuracy: 0.
6241 - val_loss: 0.9707 - val_accuracy: 0.6564
Epoch 10/40
90/90 [=====] - 40s 445ms/step - loss: 1.0360 - accuracy: 0.
6344 - val_loss: 0.9730 - val_accuracy: 0.6550
Epoch 11/40
90/90 [=====] - 42s 466ms/step - loss: 1.0087 - accuracy: 0.
6440 - val_loss: 0.9689 - val_accuracy: 0.6580
Epoch 12/40
90/90 [=====] - 42s 469ms/step - loss: 0.9855 - accuracy: 0.
6538 - val_loss: 0.8422 - val_accuracy: 0.7018
Epoch 13/40
90/90 [=====] - 39s 433ms/step - loss: 0.9700 - accuracy: 0.
6584 - val_loss: 0.9287 - val_accuracy: 0.6742
Epoch 14/40
90/90 [=====] - 42s 468ms/step - loss: 0.9461 - accuracy: 0.
6689 - val_loss: 0.8338 - val_accuracy: 0.7080
Epoch 15/40
90/90 [=====] - 41s 452ms/step - loss: 0.9301 - accuracy: 0.
6749 - val_loss: 0.8352 - val_accuracy: 0.7072
Epoch 16/40
90/90 [=====] - 39s 432ms/step - loss: 0.9149 - accuracy: 0.
6786 - val_loss: 0.8597 - val_accuracy: 0.6944
Epoch 17/40
90/90 [=====] - 43s 474ms/step - loss: 0.9099 - accuracy: 0.
6804 - val_loss: 0.8165 - val_accuracy: 0.7100
Epoch 18/40
90/90 [=====] - 42s 465ms/step - loss: 0.8818 - accuracy: 0.
6930 - val_loss: 0.7528 - val_accuracy: 0.7378
Epoch 19/40
90/90 [=====] - 41s 452ms/step - loss: 0.8762 - accuracy: 0.
6924 - val_loss: 0.8373 - val_accuracy: 0.7094
Epoch 20/40
90/90 [=====] - 39s 437ms/step - loss: 0.8636 - accuracy: 0.
6981 - val_loss: 0.7752 - val_accuracy: 0.7294
```

```

Epoch 21/40
90/90 [=====] - 42s 466ms/step - loss: 0.8514 - accuracy: 0.
7008 - val_loss: 0.7296 - val_accuracy: 0.7494
Epoch 22/40
90/90 [=====] - 42s 470ms/step - loss: 0.8467 - accuracy: 0.
7034 - val_loss: 0.7245 - val_accuracy: 0.7522
Epoch 23/40
90/90 [=====] - 42s 468ms/step - loss: 0.8367 - accuracy: 0.
7083 - val_loss: 0.7429 - val_accuracy: 0.7472
Epoch 24/40
90/90 [=====] - 41s 453ms/step - loss: 0.8236 - accuracy: 0.
7123 - val_loss: 0.7378 - val_accuracy: 0.7468
Epoch 25/40
90/90 [=====] - 41s 455ms/step - loss: 0.8128 - accuracy: 0.
7155 - val_loss: 0.7514 - val_accuracy: 0.7340

```

10.2) Evaluate Model Performance

```

In [ ]: model = tf.keras.models.load_model("Model_10")
print(f"Training accuracy: {model.evaluate(x_train_norm, y_train_split)[1]:.3f}")
print(f"Validation accuracy: {model.evaluate(x_valid_norm, y_valid_split)[1]:.3f}")
print(f"Test accuracy: {model.evaluate(x_test_norm, y_test)[1]:.3f}")

1407/1407 [=====] - 24s 16ms/step - loss: 0.6285 - accuracy: 0.7815
Training accuracy: 0.781
157/157 [=====] - 3s 17ms/step - loss: 0.7245 - accuracy: 0.7522
Validation accuracy: 0.752
313/313 [=====] - 4s 12ms/step - loss: 0.7546 - accuracy: 0.7351
Test accuracy: 0.735

```

```

In [ ]: preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)

313/313 [=====] - 4s 13ms/step
shape of preds: (10000, 10)

```

```

In [ ]: history_dict = history.history
history_dict.keys()

Out[ ]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])

```

```

In [ ]: history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

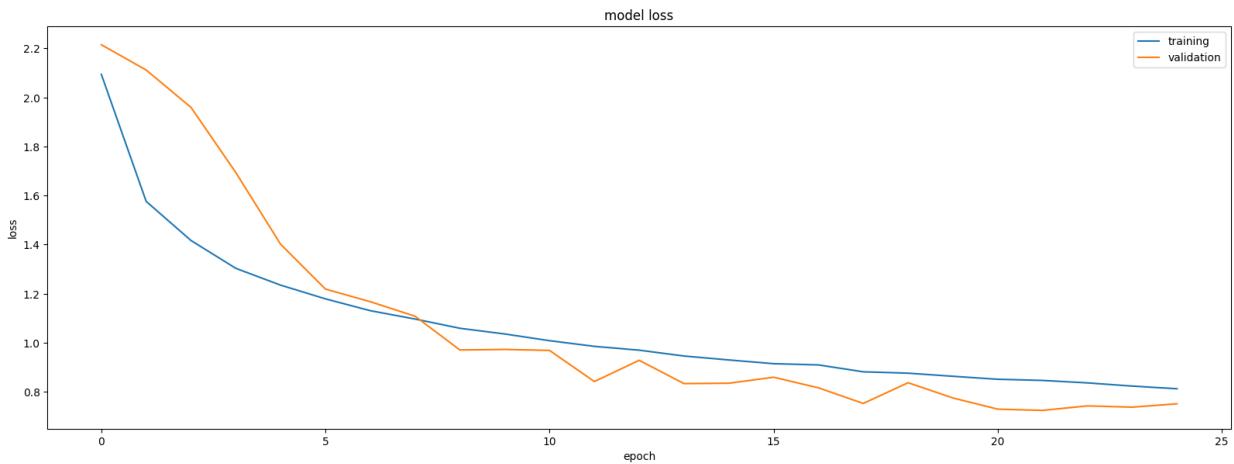
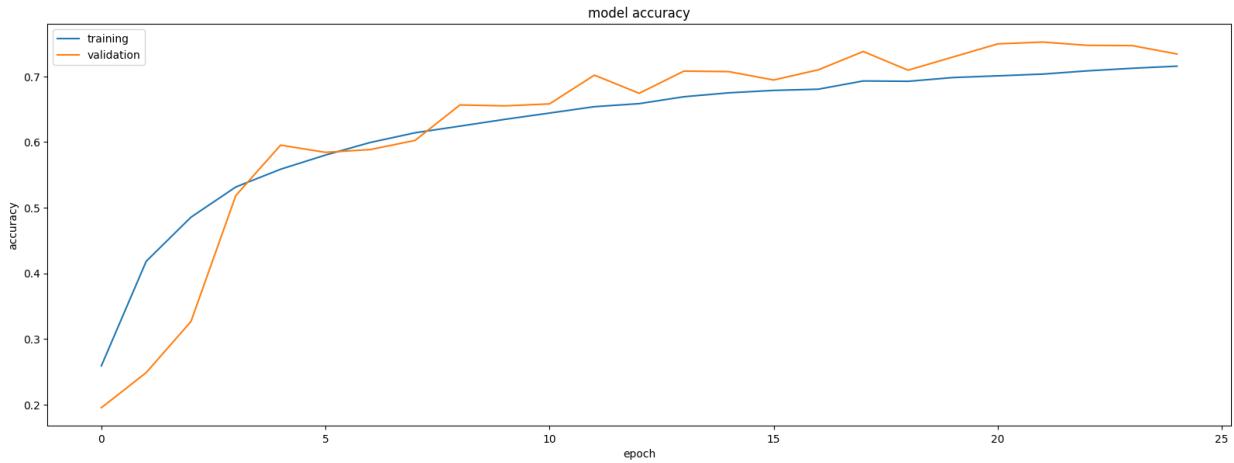
```

	loss	accuracy	val_loss	val_accuracy
20	0.851	0.701	0.730	0.749
21	0.847	0.703	0.725	0.752
22	0.837	0.708	0.743	0.747
23	0.824	0.712	0.738	0.747
24	0.813	0.715	0.751	0.734

```
In [ ]: plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```

<ipython-input-9-353fbbe40d9a>:17: MatplotlibDeprecationWarning: Auto-removal of overlapping axes is deprecated since 3.6 and will be removed two minor releases later; explicitly call ax.remove() as needed.

```
    ax = plt.subplot(subplot)
```



Let's examine the precision, recall, F1 score, and confusion matrix.

```
In [ ]: pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)
```

313/313 [=====] - 3s 9ms/step

```
In [ ]: print_validation_report(y_test, pred1)
```

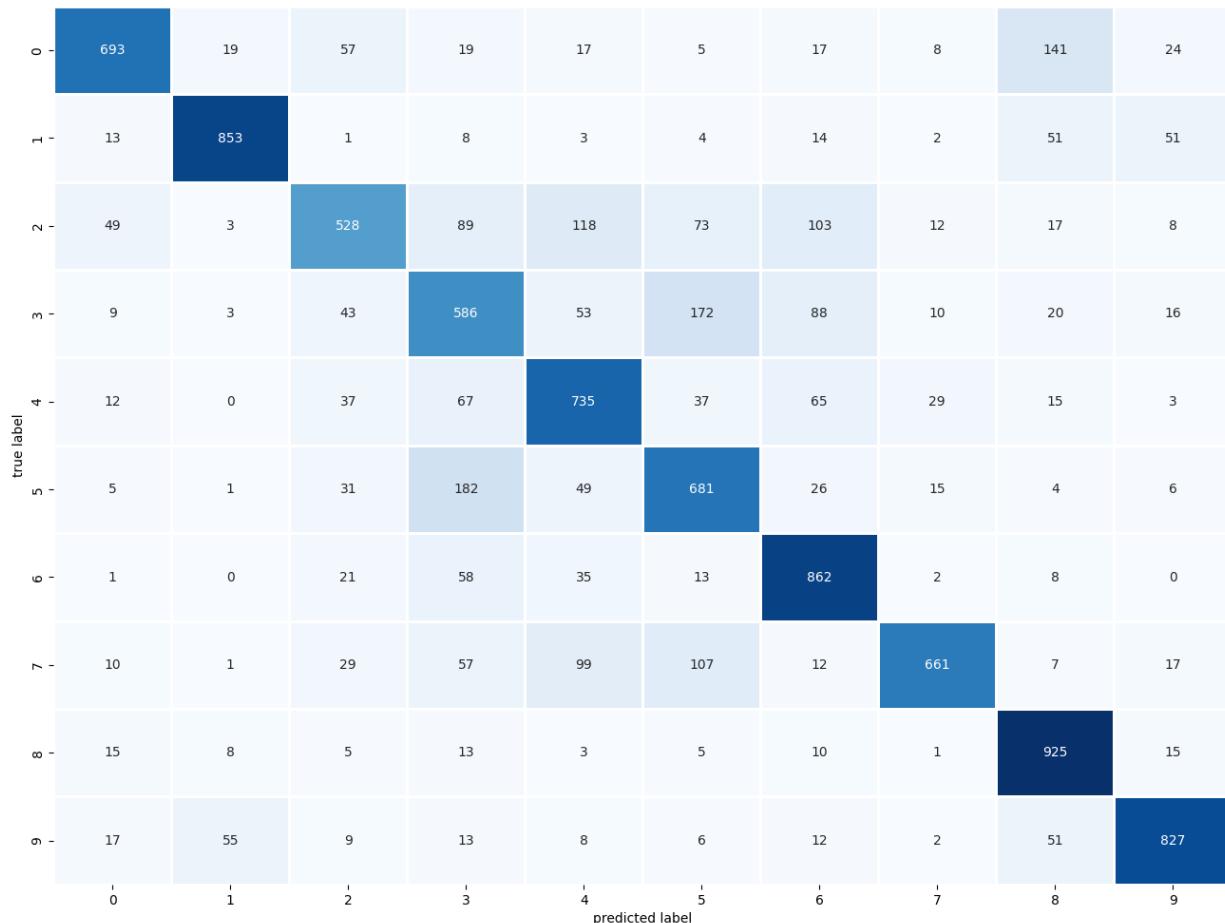
Classification Report

	precision	recall	f1-score	support
0	0.84	0.69	0.76	1000
1	0.90	0.85	0.88	1000
2	0.69	0.53	0.60	1000
3	0.54	0.59	0.56	1000
4	0.66	0.73	0.69	1000
5	0.62	0.68	0.65	1000
6	0.71	0.86	0.78	1000
7	0.89	0.66	0.76	1000
8	0.75	0.93	0.83	1000
9	0.86	0.83	0.84	1000
accuracy			0.74	10000
macro avg	0.75	0.74	0.73	10000
weighted avg	0.75	0.74	0.73	10000

Accuracy Score: 0.7351

Root Mean Square Error: 2.0494633443904284

In []: plot_confusion_matrix(y_test,pred1)



Load HDF5 Model Format

`tf.keras.models.load_model`https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model

```
In [ ]: model = tf.keras.models.load_model('Model_9')
preds = model.predict(x_test_norm)
preds.shape

313/313 [=====] - 4s 12ms/step
(10000, 10)
```

Let's examine the predictions for the testing dataset.

```
In [ ]: cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)

df = pd.DataFrame(preds[0:20], columns = [
    'airplane',
    'automobile',
    'bird',
    'cat',
    'deer',
    'dog',
    'frog',
    'horse',
    'ship',
    'truck'])

df.style.format("{:.2%}").background_gradient(cmap=cm)
```

Out[]:

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
0	2.61%	1.39%	4.94%	47.50%	0.69%	15.52%	6.09%	0.77%	19.52%	0.98%
1	4.66%	3.86%	0.01%	0.01%	0.00%	0.00%	0.00%	0.00%	91.06%	0.39%
2	7.06%	7.30%	1.06%	1.42%	0.52%	0.28%	0.52%	0.21%	77.90%	3.73%
3	52.88%	8.66%	2.23%	0.39%	0.79%	0.14%	0.24%	0.18%	32.98%	1.51%
4	0.03%	0.04%	3.36%	2.48%	11.92%	0.81%	81.23%	0.09%	0.03%	0.01%
5	0.04%	0.06%	2.69%	5.61%	3.28%	2.00%	86.02%	0.22%	0.05%	0.04%
6	4.56%	11.40%	5.72%	30.38%	1.78%	24.86%	11.05%	3.65%	1.36%	5.25%
7	0.21%	0.06%	13.06%	3.77%	27.84%	1.43%	53.23%	0.26%	0.10%	0.03%
8	0.11%	0.07%	6.86%	41.29%	5.97%	31.27%	12.57%	1.69%	0.10%	0.06%
9	2.03%	68.32%	0.24%	0.13%	0.43%	0.06%	0.49%	0.35%	1.11%	26.83%
10	25.96%	0.23%	25.37%	5.68%	29.62%	3.68%	0.45%	1.53%	7.26%	0.22%
11	0.15%	0.45%	0.00%	0.07%	0.00%	0.01%	0.01%	0.03%	2.17%	97.11%
12	0.19%	0.16%	7.85%	33.25%	10.44%	28.85%	15.90%	3.11%	0.14%	0.11%
13	0.02%	0.02%	0.31%	0.68%	2.40%	5.17%	0.00%	91.31%	0.00%	0.08%
14	1.13%	2.16%	0.06%	0.15%	0.12%	0.04%	0.06%	0.32%	5.01%	90.96%
15	4.65%	1.27%	10.81%	11.10%	10.67%	2.37%	39.86%	0.41%	18.17%	0.68%
16	0.02%	0.01%	0.31%	24.75%	0.02%	74.33%	0.03%	0.50%	0.02%	0.01%
17	1.70%	0.98%	8.65%	25.02%	16.21%	20.20%	3.62%	19.48%	0.92%	3.22%
18	1.76%	1.89%	0.01%	0.06%	0.01%	0.01%	0.02%	0.01%	86.27%	9.96%
19	0.01%	0.05%	1.75%	1.60%	5.34%	0.60%	90.53%	0.10%	0.01%	0.01%

In []:

```
(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[98]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
,'automobile'
,'bird'
,'cat'
,'deer'
,'dog'
,'frog'
,'horse'
,'ship'
,'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

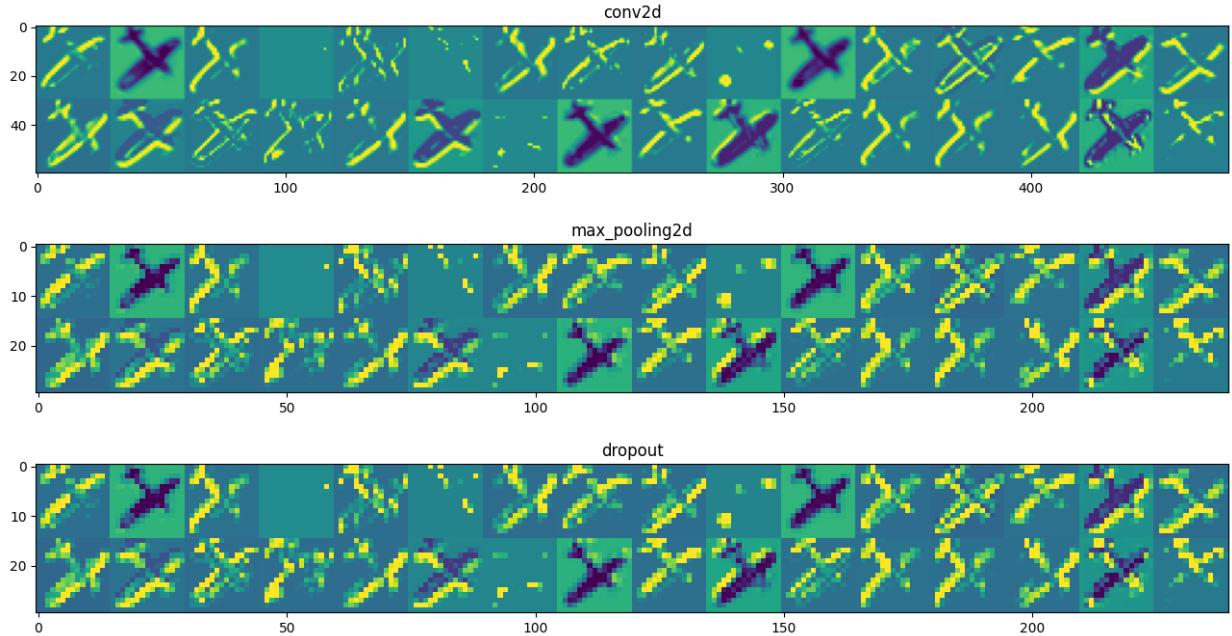
```
# Extracts the outputs of the top 8 layers:  
layer_outputs = [layer.output for layer in model.layers[:8]]  
# Creates a model that will return these outputs, given the model input:  
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)  
  
activations = activation_model.predict(img_tensor)  
len(activations)  
  
layer_names = []  
for layer in model.layers:  
    layer_names.append(layer.name)  
  
layer_names  
  
# These are the names of the layers, so can have them as part of our plot  
layer_names = []  
for layer in model.layers[:3]:  
    layer_names.append(layer.name)  
  
images_per_row = 16  
  
# Now let's display our feature maps  
for layer_name, layer_activation in zip(layer_names, activations):  
    # This is the number of features in the feature map  
    n_features = layer_activation.shape[-1]  
  
    # The feature map has shape (1, size, size, n_features)  
    size = layer_activation.shape[1]  
  
    # We will tile the activation channels in this matrix  
    n_cols = n_features // images_per_row  
    display_grid = np.zeros((size * n_cols, images_per_row * size))  
  
    # We'll tile each filter into this big horizontal grid  
    for col in range(n_cols):  
        for row in range(images_per_row):  
            channel_image = layer_activation[0,  
                                            :, :,  
                                            col * images_per_row + row]  
            # Post-process the feature to make it visually palatable  
            channel_image -= channel_image.mean()  
            channel_image /= channel_image.std()  
            channel_image *= 64  
            channel_image += 128  
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')  
            display_grid[col * size : (col + 1) * size,  
                         row * size : (row + 1) * size] = channel_image  
  
    # Display the grid  
    scale = 1. / size  
    plt.figure(figsize=(scale * display_grid.shape[1],  
                    scale * display_grid.shape[0]))
```

```
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```



1/1 [=====] - 0s 65ms/step



In []: `(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()`

```
img = test_images[122]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile'
```

```
, 'bird'
, 'cat'
, 'deer'
, 'dog'
, 'frog'
, 'horse'
, 'ship'
, 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

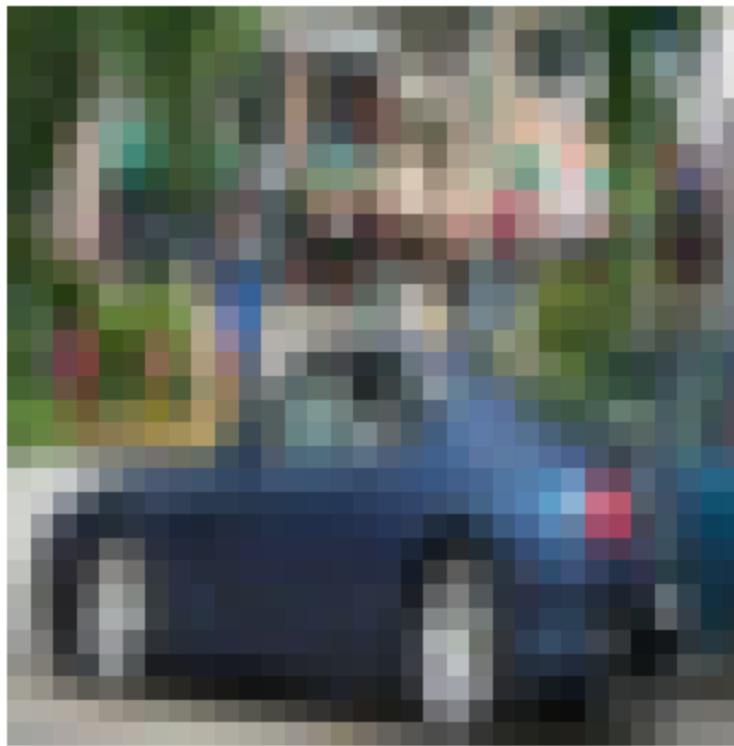
    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
```

```
col * images_per_row + row]
# Post-process the feature to make it visually palatable
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

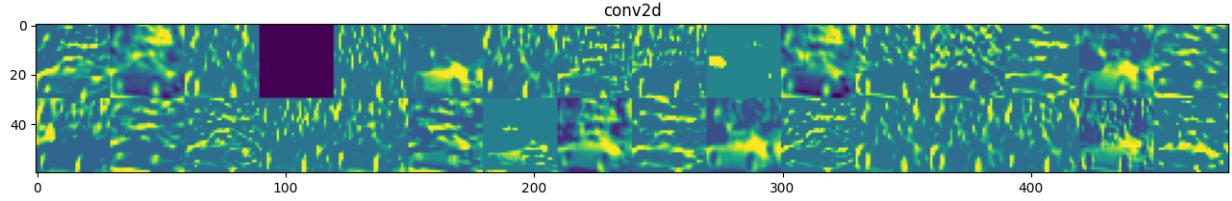
plt.show();
```

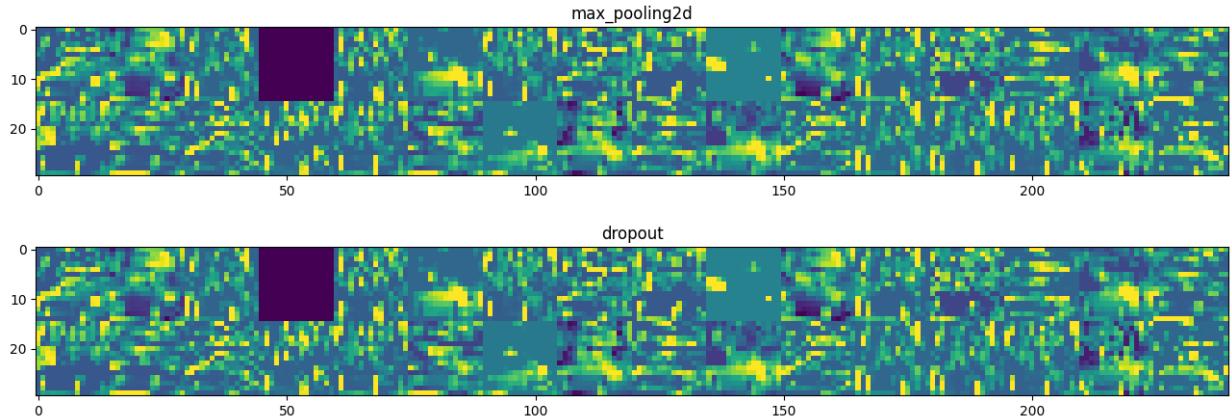


1/1 [=====] - 0s 69ms/step

<ipython-input-44-e0d043c5b9b7>:72: RuntimeWarning: invalid value encountered in divide

```
channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[75]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

These are the names of the Layers, so can have them as part of our plot

```
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

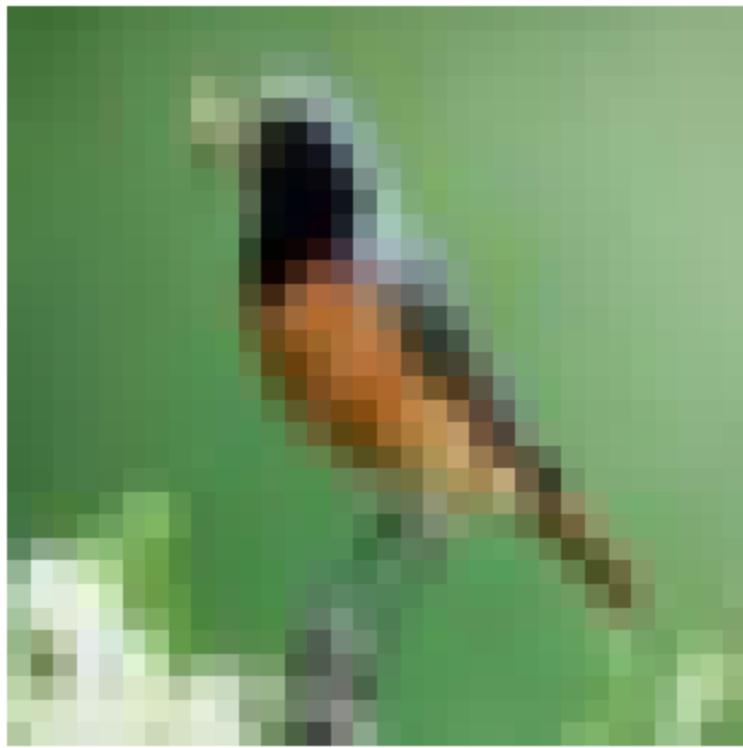
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

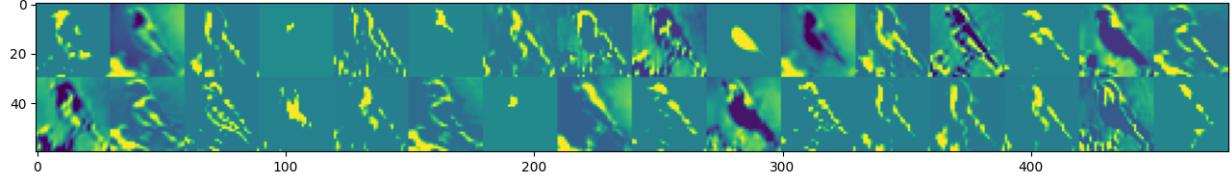
    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

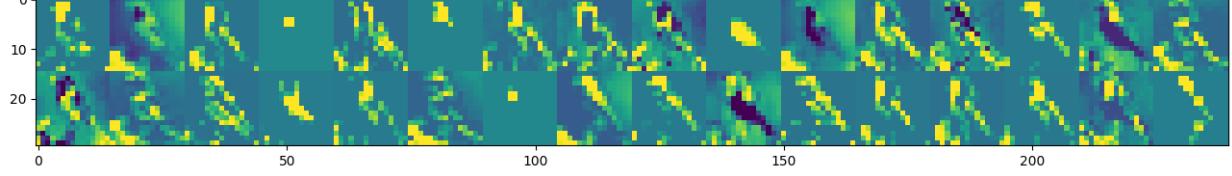


1/1 [=====] - 0s 83ms/step

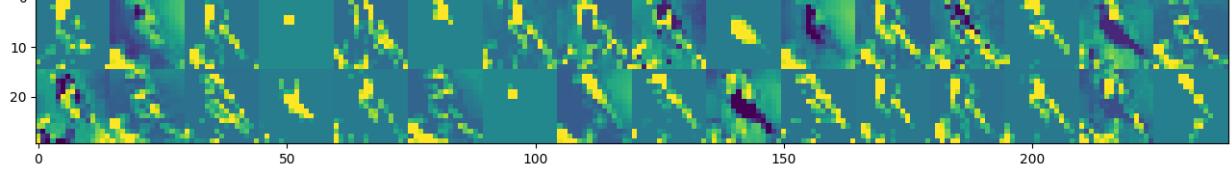
conv2d



max_pooling2d



dropout



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[184]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane'
,'automobile'
,'bird'
,'cat'
,'deer'
,'dog'
,'frog'
```

```

    , 'horse'
    , 'ship'
    , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64

```

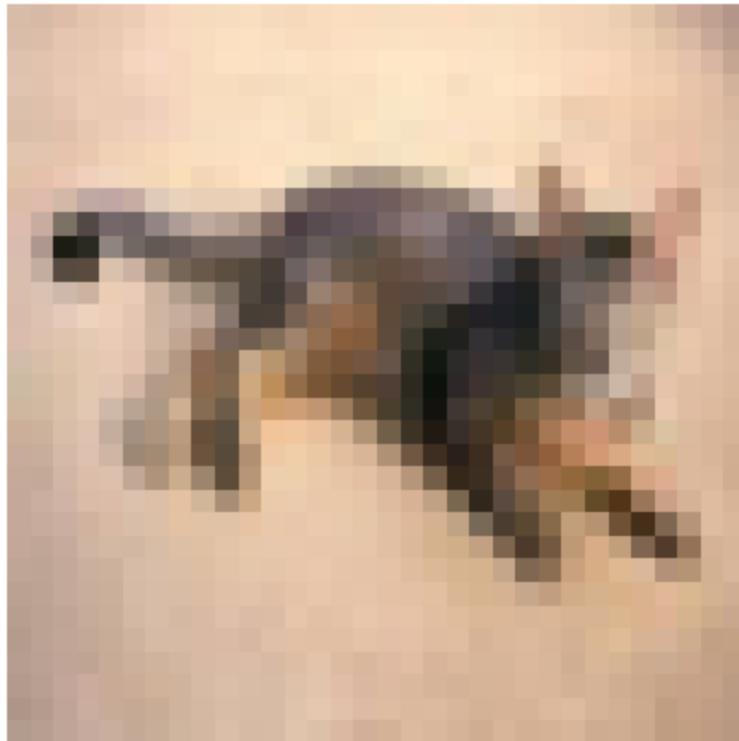
```

channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();

```

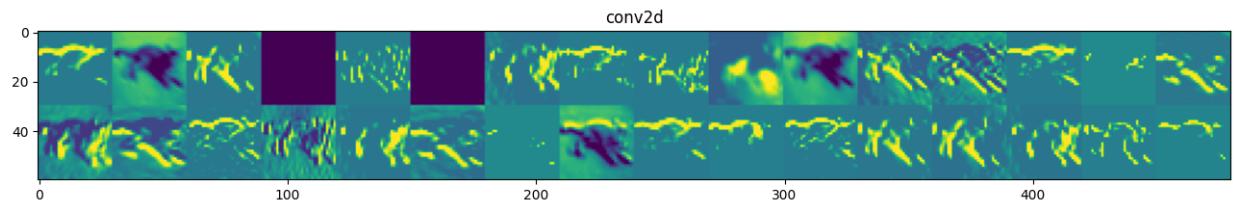


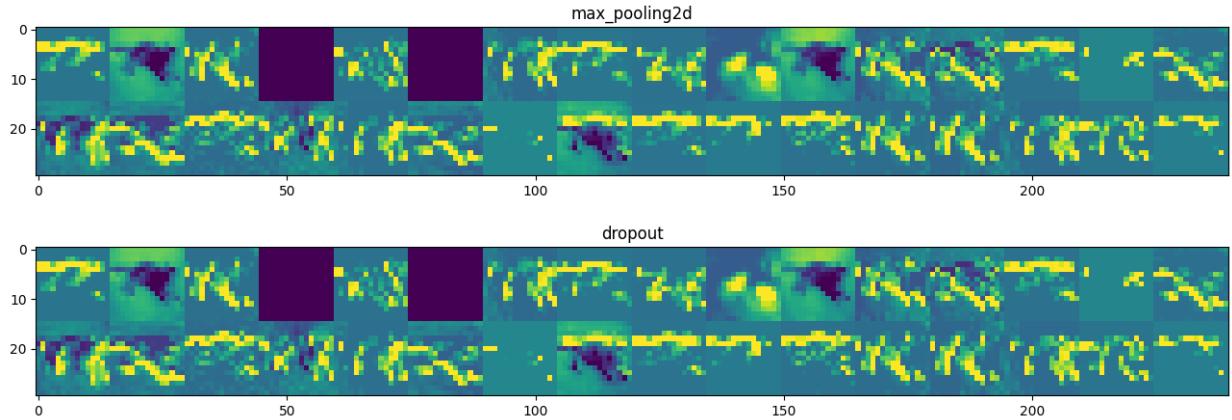
WARNING:tensorflow:5 out of the last 317 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7ceee9a9dfc0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for more details.

1/1 [=====] - 0s 74ms/step

<ipython-input-46-3bac8bdd9965>:72: RuntimeWarning: invalid value encountered in divide

```
channel_image /= channel_image.std()
```





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[159]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
```

```
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

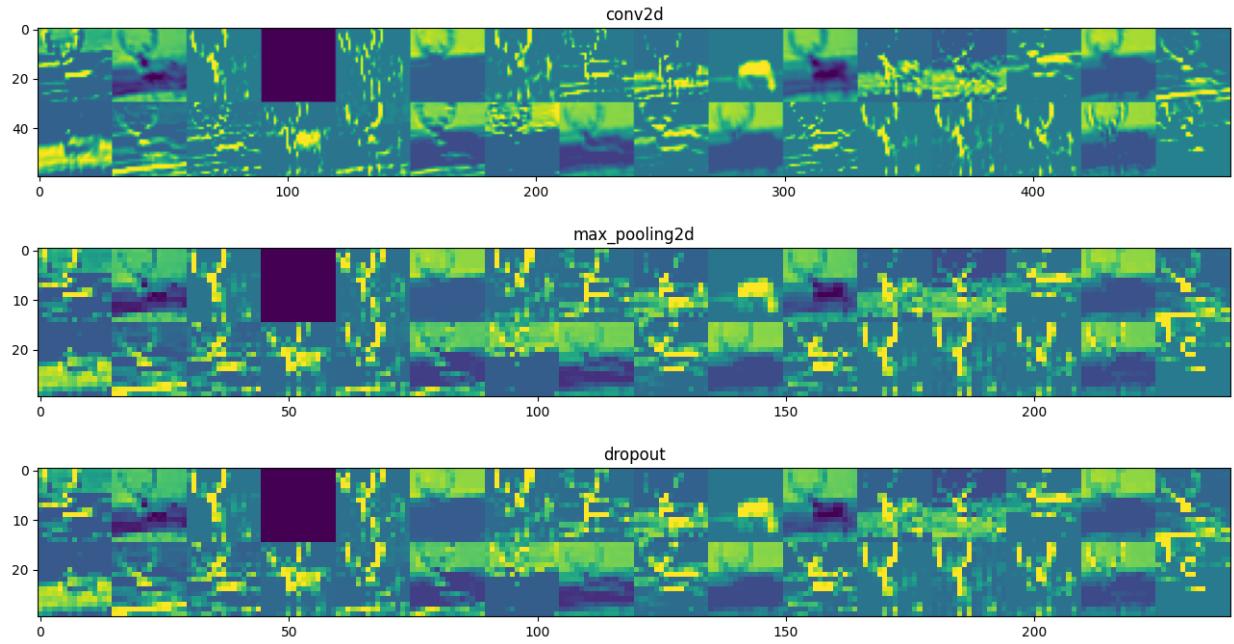
plt.show();
```



```
WARNING:tensorflow:6 out of the last 318 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7ceee9a05240> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for more details.
```

```
1/1 [=====] - 0s 82ms/step
```

```
<ipython-input-47-a52112e96c59>:72: RuntimeWarning: invalid value encountered in divide
    channel_image /= channel_image.std()
```



In []:

```
(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()

img = test_images[24]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 Layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

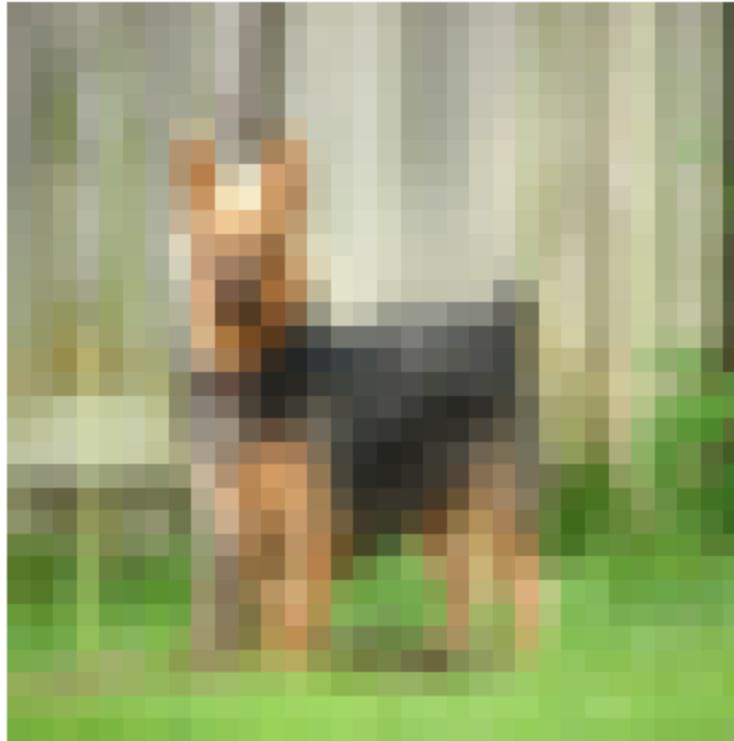
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]
```

```
# We will tile the activation channels in this matrix
n_cols = n_features // images_per_row
display_grid = np.zeros((size * n_cols, images_per_row * size))

# We'll tile each filter into this big horizontal grid
for col in range(n_cols):
    for row in range(images_per_row):
        channel_image = layer_activation[0,
                                         :, :,
                                         col * images_per_row + row]
        # Post-process the feature to make it visually palatable
        channel_image -= channel_image.mean()
        channel_image /= channel_image.std()
        channel_image *= 64
        channel_image += 128
        channel_image = np.clip(channel_image, 0, 255).astype('uint8')
        display_grid[col * size : (col + 1) * size,
                     row * size : (row + 1) * size] = channel_image

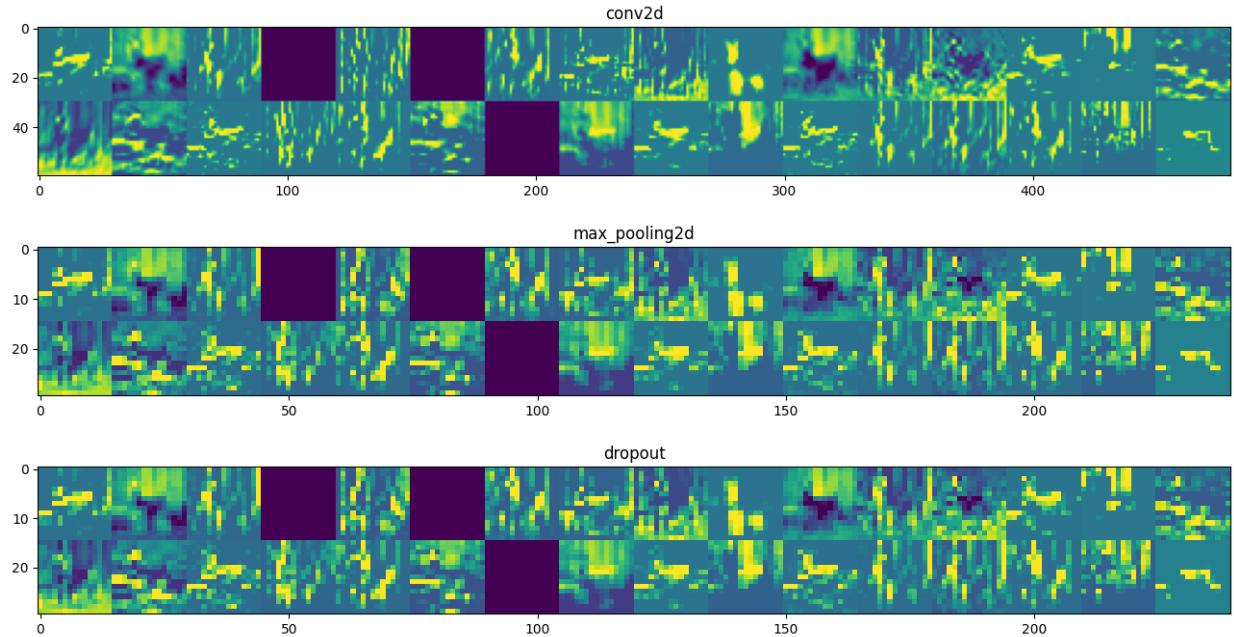
# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```



1/1 [=====] - 0s 75ms/step

<ipython-input-48-edee34e9d17b>:72: RuntimeWarning: invalid value encountered in divide
 channel_image /= channel_image.std()



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[152]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[:, :, :, col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

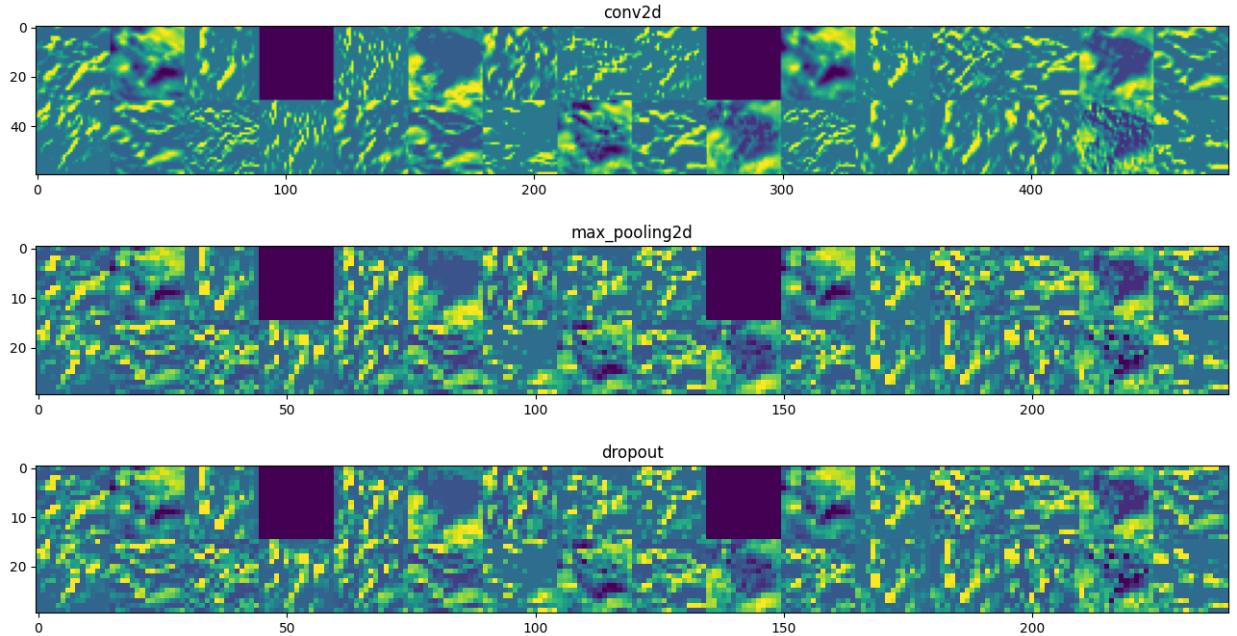
plt.show();
```



```
1/1 [=====] - 0s 71ms/step
```

```
<ipython-input-49-0ef3f9c61bfb>:72: RuntimeWarning: invalid value encountered in divide
```

```
    channel_image /= channel_image.std()
```



```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[2004]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat']
```

```

        , 'deer'
        , 'dog'
        , 'frog'
        , 'horse'
        , 'ship'
        , 'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

activations = activation_model.predict(img_tensor)
len(activations)

layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

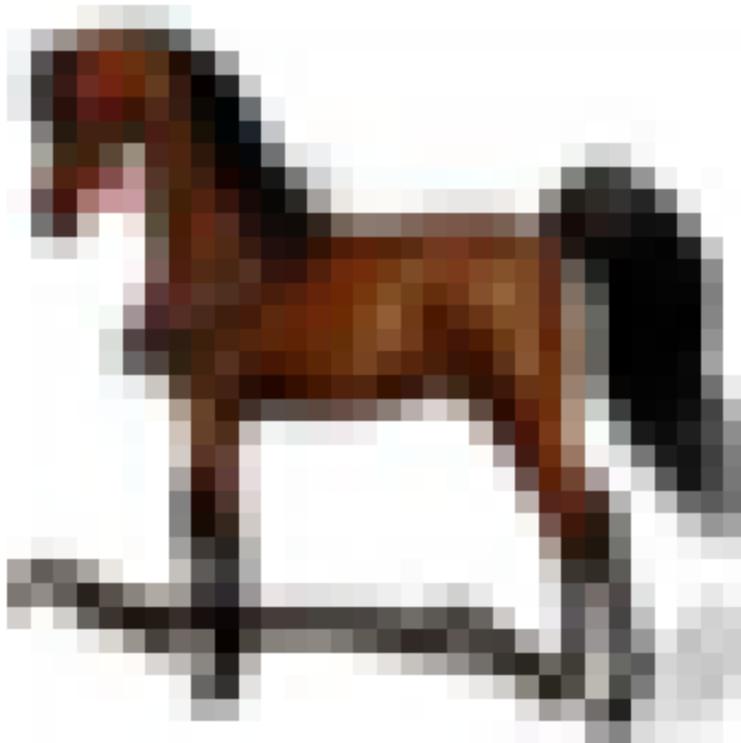
    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable

```

```
channel_image -= channel_image.mean()
channel_image /= channel_image.std()
channel_image *= 64
channel_image += 128
channel_image = np.clip(channel_image, 0, 255).astype('uint8')
display_grid[col * size : (col + 1) * size,
             row * size : (row + 1) * size] = channel_image

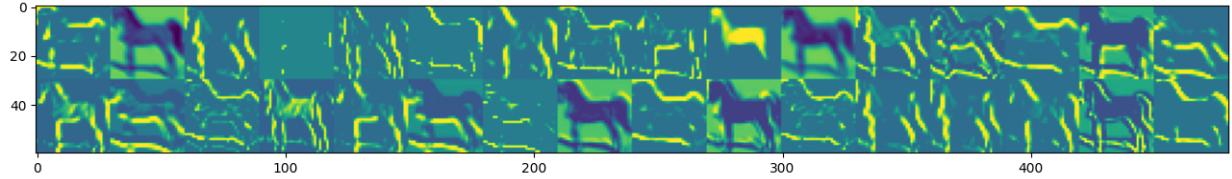
# Display the grid
scale = 1. / size
plt.figure(figsize=(scale * display_grid.shape[1],
                   scale * display_grid.shape[0]))
plt.title(layer_name)
plt.grid(False)
plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

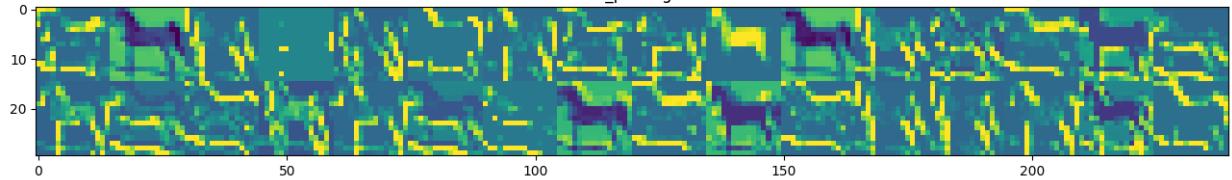


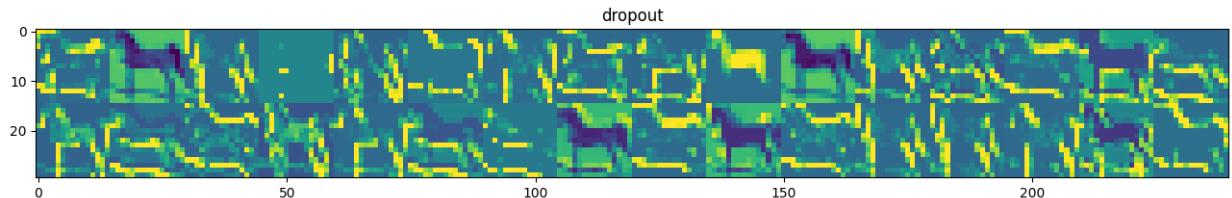
1/1 [=====] - 0s 70ms/step

conv2d



max_pooling2d





```
In [ ]: (_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
```

```
img = test_images[185]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
               'automobile',
               'bird',
               'cat',
               'deer',
               'dog',
               'frog',
               'horse',
               'ship',
               'truck']

plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

```
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)
```

```
layer_names
```

```
# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now Let's display our feature maps
```

```
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

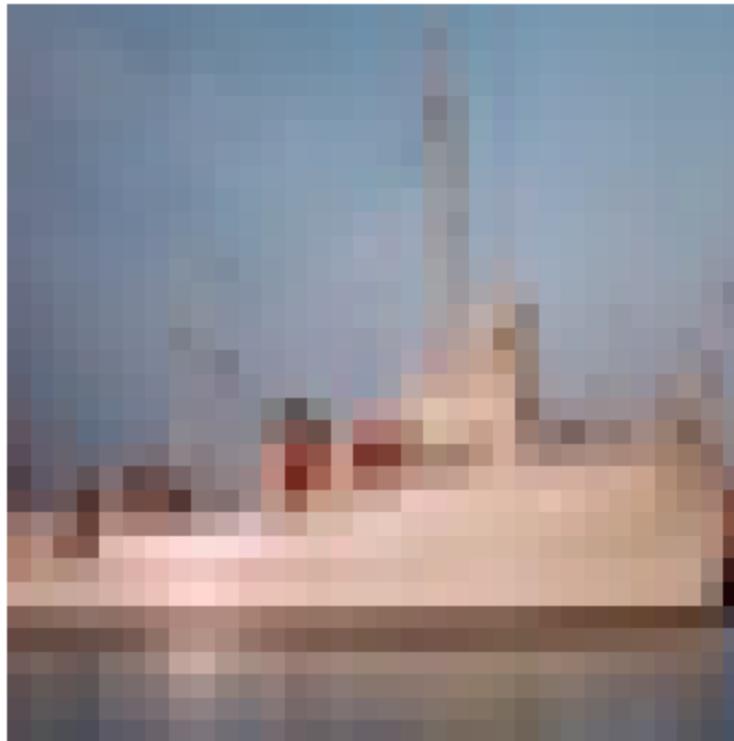
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

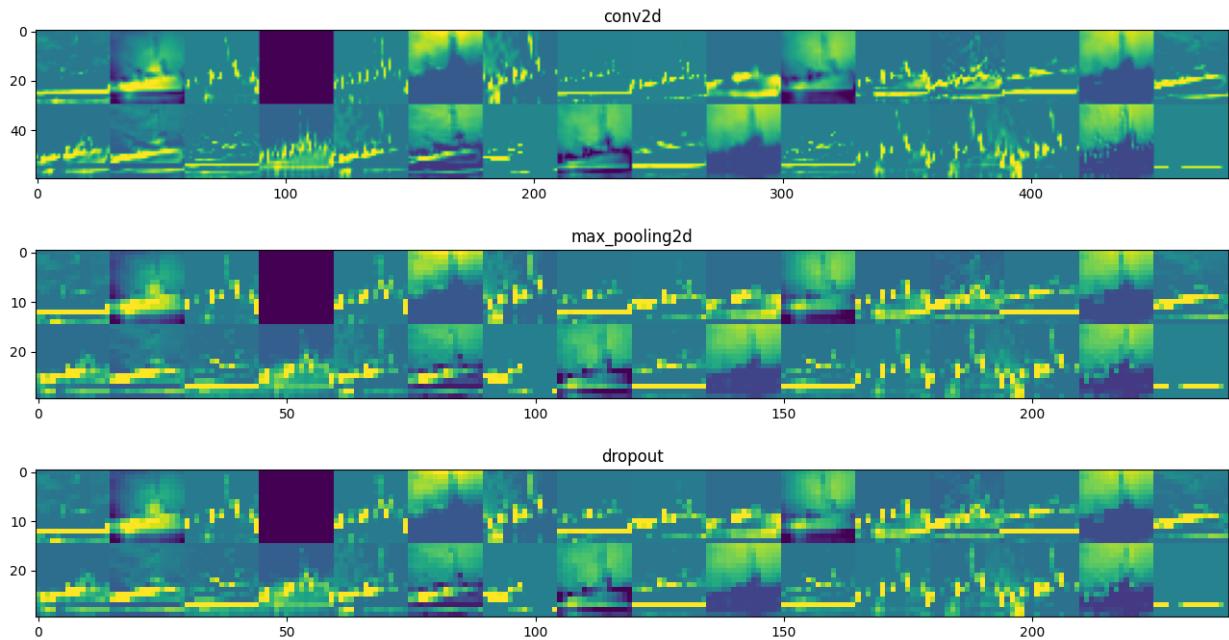
plt.show();
```



1/1 [=====] - 0s 77ms/step

<ipython-input-51-f532589aadeb>:72: RuntimeWarning: invalid value encountered in divide

channel_image /= channel_image.std()

In []: `(_,_), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()`

```
img = test_images[133]
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

class_names = ['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

```
plt.imshow(img, cmap='viridis')
plt.axis('off')
plt.show()
```

Extracts the outputs of the top 8 layers:

```
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
activations = activation_model.predict(img_tensor)
len(activations)
```

```
layer_names = []
for layer in model.layers:
    layer_names.append(layer.name)

layer_names

# These are the names of the Layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:3]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

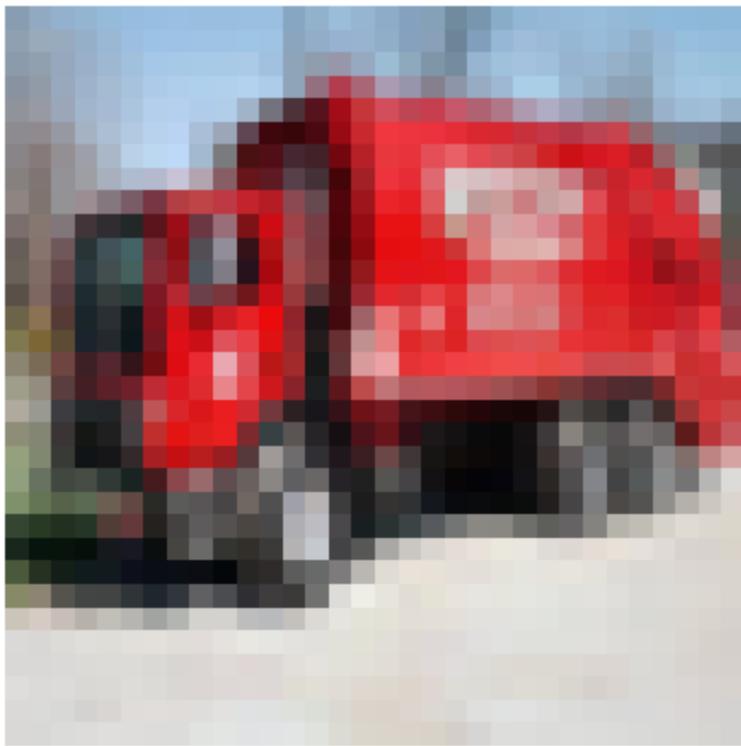
    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

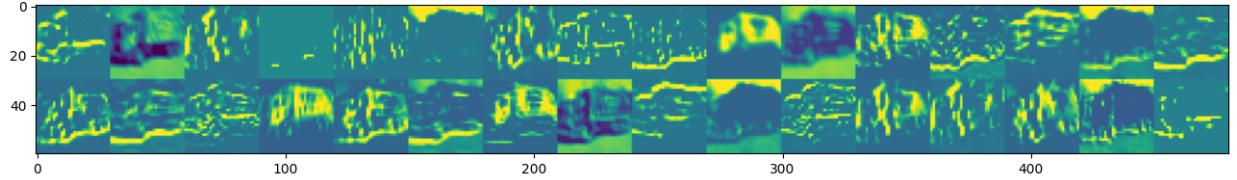
    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                      scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show();
```

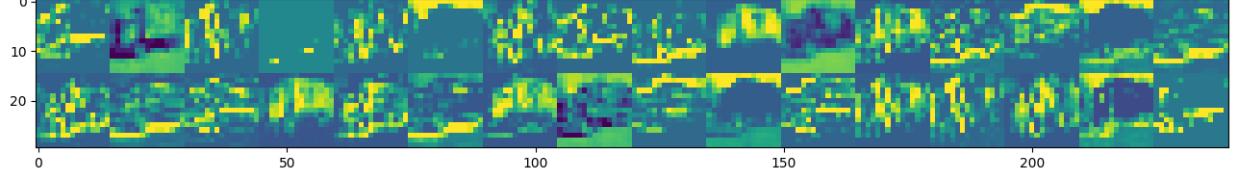


1/1 [=====] - 0s 81ms/step

conv2d



max_pooling2d



dropout

