Advanced Machine Learning Block 3

Custom CNN Open Project

Maria Paraskeva and Omar López Rubio

Universitat Politécnica de Catalunya, Facultat d'Informàtica de Barcelona

January 2024

ABSTRACT

Context. To address five distinct tasks spanning model architecture, experimentation, interpretability, and transfer learning using three different data sets to create custom Convolutional Neural Networks (CNN).
Aims. Evaluating the memory requirements and computational load of different layers in a custom CNN model, conducting experiments to assess the impact of various factors on model performance, visualizing and interpreting CNN activations and filters, exploring transfer learning, and culminating in an open project with extensive data exploration, model improvement, and interpretability analysis.
Methods. Variety of methods for Deep Neural Networks including resampling methods (SMOTE), dimensionality reduction methods (PCA), transfer learning, regularization techniques, preprocessing methods (whitening, data augmentation).
Results. Observe how neural networks capture data with different layers. Pre-trained models underfit seemingly potent networks. Satellite image datasets may be difficult for CNNs as proven by various relevant works.

Key words. CNN - Deep Learning - CIFAR-10 - Terrassa 900 - Satellite image Classification Dataset-RSI-CB256

1. Introduction

The development and optimization of Convolutional Neural Networks (CNNs) have become integral to addressing a variety of tasks in computer vision, ranging from image classification to object detection. In this assignment of the Advanced Machine Learning course we are going to use a various different techniques and datasets in order to implement the methods and objectives seen in class as well as try new ones and see the results.

2. Custom CNN

Using the CIFAR-10 dataset we created a basic CNN model that consists of the following layers: Conv2D, MaxPooling2D, Flatten, and two Dense layers. In order to study the memory requirements and computational load of each layer, it is essential that we examine them individually.

2.1. Conv2D Layer

The first layer is the Conv2D layer which performs a 2D convolution operation in the input data. Since our dataset comprises of images, this layer slides a set of filters over the input image and computes a dot product at each step. The weights of these filters are called kernels. The result is a set of feature maps that capture different aspects of the input, called a tensor of outputs, which can then be used as an input of a new convolutional layer.

It is obligatory to set the number of filters that the convolutional layers will learn from, determining at the same time how many output filters there will be in the convolution. Our model has 32 filters with kernel dimensions of size 3×3, making each filter a 3×3 matrix. The kernel size specifies the height and width of the 2D convolutional window and must be an odd integer in order to maintain a center pixel and preserve symmetry while detecting image patterns. An additional dimension (x3) is added as each pixel is an RGB vector. Also, there is an added bias term to the weighted sum before applying the filters, which introduces flexibility and additional capacity to the model, allowing it to better capture complex the patterns.

So, the dimensions of the data produced by this layer, or, the output shape, are 32 feature maps of size 30x30 which have no specific batch size, shown as "None" in the model summary.

The number of parameters is calculated as follows:
(Filter Height * Filter Width * Input Channels + 1 bias term) * Number of Filters = (3 * 3 * 3 + 1) * 32 = 896 total parameters.

Regarding memory requirements, we can say that they are relatively low compared to the dense layers that we add later in our model, but the computational load is higher due to the convolution operations. During convolutions, a significant number of multiplications and additions are involved, raising the computational cost.

2.2. MaxPooling2D Layer

The next layer in our model is a max pooling layer, used to lower spatial dimensions (height and width) by reducing the dimensions of the feature maps by half so as to help save memory. This is achieved by dividing the input feature maps into non-overlapping regions and selecting the maximum value from each region to then form the output. The pooling window we chose is of size 2×2.

As the default stride for MaxPooling2D is typically set equal to the size of the pooling window, the stride here is set to 2. This means that the window moves by two steps horizontally and two steps vertically, reducing the spatial dimensions by half.

There are no learnable parameters in this layer and, since the dimensions are "cut" in half, the output shape of this layer is a series of 32 feature maps of size 15×15 with no fixed batch size just like the previous layer. The number of channels remains the same (32).

As a result, max pooling helps with reducing the computational load for subsequent layers and also in controlling overfitting. The required memory in this layer is for storing the output feature map, which is smaller than the input due to pooling. The computational load is relatively lower compared to convolutional layers, involving selecting the maximum value from each 2×2 block.

2.3. Flatten Layer

The objective of this layer is to convert the 3D output from the previous layer into a 1D array and, thus, preparing the data for the dense layer that follows. This operation does not introduce additional computational requirements but just reshapes the data. Similarly to the previous layer, there are no learnable parameters either. Depending on the size of the feature map, the memory requirements for storing the flattened data can be significant. In our case, the computations that take place in this step are:
Number of Elements in a Channel * Number of Channels = (15 * 15) * 32 = 225 * 32 = 7200 values per image.

The number of elements is in each channel is the product of the spatial dimensions.

2.4. Dense Layer (Fully connected)

Finally we added two dense layers in our model, one fully connected and the next one that will give us the final output. Fully connected means that every neuron of the previous layer is connected to every neuron of the current layer. We decided to add 64 neurons that will fully connect to the 7200 neurons from the Flatten layer explained above. The primary parameters in this layer are weights and biases, as each connection between neurons has an associated weight, and each neuron has a bias term. The weights are organized into a matrix. If a dense layer has n neurons and the preceding layer has m neurons, the weight matrix will have dimensions $m \times n$. Regarding biases, each neuron in the dense layer has its own bias term, contributing to the flexibility and expressiveness of the layer. In our case we are dealing with the following number of parameters:
Number of Inputs * Number of Neurons + Number of Biases = 7200 * 64 + 64 = 460,864 total parameters.

Evidently, there are high memory requirements for storing the weights and biases. Also, this layer is computationally intensive due to the matrix multiplication operations.

Apart from that, an activation function is typically applied to the weighted sum of inputs and biases in each neuron. We are using a common activation function called ReLU (Rectified Linear Unit), which introduces non-linearity to the model, allowing the network to learn and represent complex patterns in the data. From this choice we aimed to benefit from its sparse activation and the fewer vanishing gradient problems [(**?**)] compared to other activation functions like sigmoid.
The output of this dense layer is obtained by applying the activation function to the weighted sum of inputs and biases. This layer contributes to the hierarchical learning of features and patterns in the data. The result is a non-linear transformation of the input data that has a shape determined by the number of neurons which, in our case, are 64 in total.

2.5. Dense Layer (Output)

The second dense layer which is the last one in our model is a fully connected dense layer that has 10 neurons. Each of these 10 neurons are respectively connected to all 64 neurons from the previous layer, producing the classification probabilities for the 10 classes in the CIFAR-10 dataset. There are weights and biases as well, with the weight matrix measuring at 64×10 dimensions and the biases being 10, one for each neuron in the current layer. The computational cost is similar to the previous dense layer, but with fewer parameters due to fewer output neurons. Hence, the calculation for the number of parameters is: Number of Inputs * Number of Neurons + Number of Biases = 64 * 10 + 10 = 650 total parameters and an output shape of 10 neurons. It is evident that there is a lower need for memory requirements compared to the first dense layer.
The activation function used here is the softmax function. Softmax is commonly employed in the output layer of a classification model, and it converts the raw output scores (logits) of the network into probabilities. The softmax function exponentiates the scores by applying the standard exponential function to each input element and normalizes them to ensure that they sum to 1, providing a probability distribution over the classes.

2.6. Discussion

Exploring the differences between layer types, we notice that Conv2D layers have fewer parameters due to shared weights and local connectivity but require more computations for convolution operations. They mainly focus on feature extraction with shared weights. Meanwhile, dense layers have more parameters (hence more memory usage) as each neuron is connected to all neurons in the previous layer, but the computations are straightforward matrix multiplications. The max pooling layer reduces the dimensionality of the data, decreasing the computational load and memory requirements for subsequent layers. Lastly, the flatten layer has no computational cost or additional parameters but impacts memory due to the potential large size of the flattened vector as it reshapes data for transitioning between convolutional and dense layers.

Apart from the different layers, there are more decisions to be taken when creating a convolutional neural network model. One of them is the optimizer used, which is responsible for updating the weights of the neural network based on the computed gradients. In this case, the optimizer chosen is Adam, a popular optimization algorithm. The *learningrate* = 0.001 argument sets the learning rate, which determines the size of the steps taken during optimization.

The loss function is another measure in a CNN model which measures how well the model is performing. For categorical classification tasks, where the goal is to predict one out of multiple classes, *categorical crossentropy* is a commonly used loss function. It quantifies the difference between the predicted probabilities and the true class labels.

Lastly, metrics are used to evaluate the performance of the model during training and testing. We chose the *accuracy* metric which is a commonly used metric for classification tasks. It represents the proportion of correctly classified samples.

Putting it all together, we have our final model which, during training, uses the selected optimizer to adjust the weights in order to minimize the chosen loss function, and the accuracy metric is monitored to assess the model's performance.

3. Experiments

In this part of our analysis we are going to use the model created in the previous section and study the impact of tweaking the model's hyperparameters or adding new ones. First we are going to implement data augmentation, then play with the size of the training batch, and add batch normalization. Finally, we are going to force overfitting of the data and then solve it with some regularization techniques.

3.1. Data Augmentation

Data augmentation is used to increase the diversity of the training data by introducing variations in the images. During training, the image generator creates augmented versions of the original images on-the-fly as the model is trained. Each batch of training data will contain a mix of the original images and their augmented versions, which helps the model generalize better and reduces the risk of overfitting to the training data. In order to apply data augmentation to our input images during the training process, we used the basic *ImageDataGenerator* function from Keras. The parameters used were the following:

1. rotation range, to specify the range (in degrees) within which random rotations can be applied to the input images. In this case, it is set to 20, which means that the generator may randomly rotate images by up to 20 degrees clockwise or counterclockwise.
2. width shift range and height shift range, to control the range (as a fraction of the total width or height) within which random horizontal and vertical shifts can be applied to the images. We set both of them to 0.2. which means that the generator can shift the image horizontally as well as vertically by up to 20% of its total width.
3. horizontal flip, which when set to True, enables random horizontal flips (mirror flips) of the images. This means that the generator may flip some of the images horizontally with a 50% chance. This can be useful to create additional training samples for symmetric objects.
4. zoom range, to control the range for random zooming. A value of 0.2 indicates that the generator can randomly zoom in or out on the images by up to 20%. This helps the model learn to recognize objects at different scales.

The results of this implementation in the data can be seen in Figure 1 as well as Table 1. The figure shows that the null model has an higher curve in accuracy in both train and validation data sets, indicating overfitting. Taking a look at the table we see that improvement in training accuracy is observed for both models, but the null model starts with a higher initial accuracy and achieves a larger increase, similar for validation accuracy, too. Data augmentation appears to have helped the augmented model narrow the performance gap with the null model. Both models experience a reduction in the loss metric, with the null model showing a more substantial decrease. In general, data augmentation has had a positive effect on the models' ability to generalize to the training data but may not necessarily result in improved generalization to the validation data in terms of loss. Regarding the computation time, both models had the same duration in total, with 4-6ms/step and 1250 steps per epoch.

Table 1. Accuracy and Loss Comparison with Data Augmentation.

Metric	Null model	Augmented model
Train Accuracy	0.40 → 0.64	0.20 → 0.37
Val. Accuracy	0.47 → 0.58	0.25 → 0.31
Train Loss	1.66 → 1.01	2.09 → 1.68
Val. Loss	1.47 → 1.20	1.96 → 1.87

Table 2. Training and Validation Accuracy and Loss in different Batch Sizes.

Metric	Batch=16	Batch=32	Batch=64
Train Accuracy	0.25 → 0.33	0.25 → 0.36	0.25 → 0.36
Val. Accuracy	0.30 → 0.33	0.29 → 0.31	0.27 → 0.30
Train Loss	1.98 → 1.79	1.99 → 1.74	2.01 → 1.72
Val. Loss	1.89 → 1.91	1.92 → 2.00	2.01 → 2.04

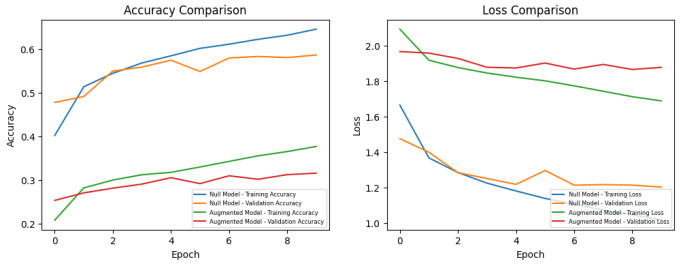

Fig. 1. Accuracy and Loss Comparison Plots with Data Augmentation.

3.2. Size of training batch

When it came to the size of the training batch, we decided to try three different values: 16, 32, and 64. Changing the size refers to altering the number of training examples that are processed together in each iteration. The size of the training batch can have a significant impact on the training dynamics and performance of the model.

Looking at Table 2, we notice that larger batch sizes (Batch=32 and Batch=64) achieve higher final training accuracy compared to Batch=16, suggesting that larger batch sizes allow the model to learn more from each batch and converge to a better solution on the training data. The validation accuracy, which measures the model's ability to generalize to unseen data, is highest for Batch=32, indicating that this batch size may result in the best generalization. In addition, the two larger batch sizes lead to lower final training losses, indicating better fitting to the training data, while the smaller size has a slightly higher final training loss. However, validation loss increases for all three models showing that it is possible for the models to overfit. Figure 2 shows that the two highest batch sizes have a similar high training accuracy, confirming our speculations. Overfitting may occur because generally smaller batch sizes (the usual range being from 32 to 128) have higher noise in gradient updates, which can lead to a less stable convergence.

Regarding computation time, the steps of the different models were 3125, 1563, and 782 respectively, decreasing in number as the batch sizes increased. However, that didn't have a huge impact in total computation time as our dataset is quite small, with each epoch ranging from 34 to 46 seconds for all three models. The lowest execution time was observed in the model with batch size = 64.

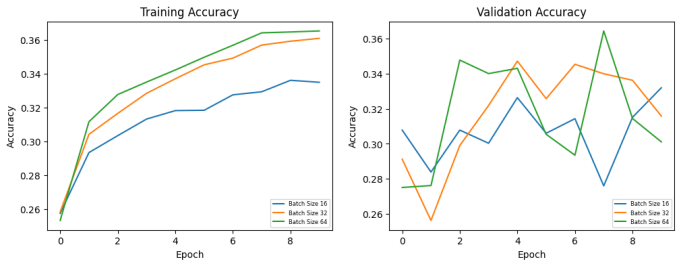

Fig. 2. Training and Validation Accuracy in different Batch Sizes.

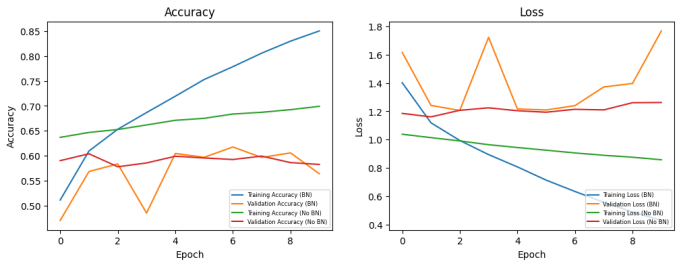

Fig. 3. Accuracy and Loss comparison plots with Batch Normalization.

Table 3. Accuracy and Loss comparison with Batch Normalization.

Metric	Null model	BatchNorm model
Train Accuracy	0.63 → 0.69	0.55 → 0.85
Val. Accuracy	0.59 → 0.58	0.47 → 0.56
Train Loss	1.03 → 0.85	1.40 → 0.42
Val. Loss	1.18 → 1.26	1.61 → 1.76

3.3. Batch normalization

Batch Normalization (BN) is a technique used in order to improve the training stability and convergence speed of the model and it is applied to the activations of a neural network layer, typically before the activation function. In each mini-batch during training, BN normalizes the activations of the layer by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. This helps center and scale the activations. After normalization, BN applies two learnable parameters, typically referred to as gamma (γ) and beta (β), which allow the network to scale and shift the normalized activations. In our implementation of batch normalization we did not provide specific values for these two parameters, hence giving the layer the freedom to learn them during training. This way the network will adaptively adjust the mean and scale of the activations during training based on the data. BN also introduces a small amount of noise or randomness to the training process, which can help improve model generalization and stabilize training. It reduces the internal covariate shift, which is the change in the distribution of activations as the model trains.

Table 3 shows the resulting metrics of both models. The training accuracy with BN starts lower but shows significant improvement throughout training, showing that it tends to stabilize and accelerate training, which is reflected in the faster improvement in training accuracy and lower training loss early on. However, in the BN case, the model appears to overfit the training data, as evidenced by the increasing validation loss and limited improvement in validation accuracy. In the case without BN, the model performs better in terms of generalization to the validation data, as indicated by the stable validation accuracy and lower validation loss. A good solution would be to use the model with batch normalization and try to tackle overfitting, case that is evident following the blue line in Figure 3

3.4. Overfitting

In the last part of this task we had to force overfit and then solve it using methods that control the complexity of the model. We took our initial model used in the previous tasks and added extra layers that would result in a model with a higher capacity, making it more prone to overfitting. Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise or random fluctuations, leading to poor generalization on new, unseen data.

The resulting complex model was a sequential neural network, implying a linear stack of layers. It begins with two convolutional layers (Conv2D) with 32 filters each, a kernel size of (3, 3), Rectified Linear Unit (ReLU) activation, 'he_uniform' kernel initializer, and 'same' padding. These layers are followed by a max-pooling layer (MaxPooling2D) with a pool size of (2, 2). The pattern is repeated with two more pairs of convolutional layers and max-pooling layers, gradually increasing the number of filters to 64 and then 128. The final convolutional layer is followed by flattening the output and two dense (fully connected) layers. The first dense layer has 128 neurons with ReLU activation and 'he_uniform' kernel initializer, and the second dense layer has 10 neurons with softmax activation, suitable for a multi-class classification problem with 10 classes. The network's architecture can be seen in detail in Figure 5.

The model was compiled using the Adam optimizer with a learning rate of 0.001, categorical crossentropy loss, and accuracy as the evaluation metric. Regarding training, the model was trained for 50 epochs and the training process was verbose, meaning it provided detailed output during training that helped us compare it to the simplified version we developed below.

The deliberate overfitting can be inferred from the high capacity of the model (many parameters) and the absence of regularization techniques like dropout or batch normalization. Table 4 shows the accuracy and validation scores in both training and validation datasets and the substantial improvement in training accuracy from 50% to 96% as well as the reduction in training loss from 1.37 to 0.10 highly indicate overfitting. In the meantime, the improvement in validation accuracy and the increase in validation loss are more limited. The model appears to be memorizing the training data but struggles to generalize to new examples.

Once we successfully induced overfitting, the next step involved applying methods to mitigate this issue. The simplified version of the model starts with a convolutional layer (Conv2D) with 32 filters, a kernel size of (3, 3), ReLU activation, 'he_uniform' kernel initializer, 'same' padding, and an input shape of (32, 32, 3) indicating the input image dimensions. Batch normalization (BatchNormalization) is applied after each convolutional layer, which helps stabilize and accelerate training by normalizing the input to a layer. Max-pooling (MaxPooling2D) with a pool size of (2, 2) was used to downsample the spatial dimensions. Dropout (Dropout) is applied after the max-pooling layer, introducing regularization by randomly setting a fraction of input units to zero during training. This pattern is repeated with additional pairs of convolutional layers, batch normalization, max-pooling, and dropout, gradually increasing the

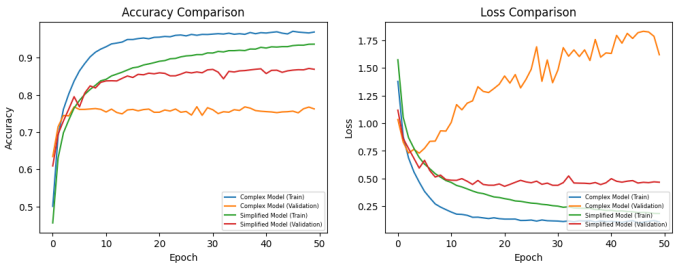

Fig. 4. Accuracy and Loss in Complex and Simplified models.

Table 4. Accuracy and Loss Comparison with Complex & Simplified models.

Metric	Complex model	Simplified model
Train Accuracy	0.50 → 0.96	0.45 → 0.93
Val. Accuracy	0.63 → 0.76	0.60 → 0.86
Train Loss	1.37 → 0.10	1.57 → 0.18
Val. Loss	1.03 → 1.61	1.11 → 0.46

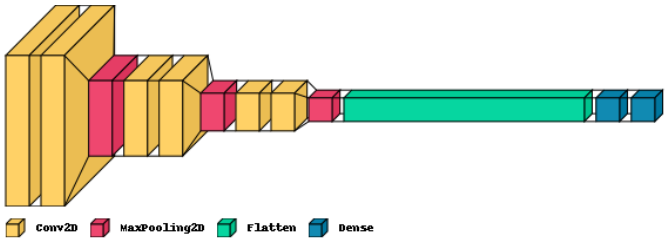

Fig. 5. CNN architecture of the Complex model.

number of filters to 64 and then 128. After the convolutional layers, we added a flattening layer to convert the 3D output to 1D, followed by two dense (fully connected) layers. Batch normalization and dropout are also applied after the first dense layer, as can be seen in Figure 6.

The model was compiled using the same methods as its complex version, and also trained for the same amount of epochs using the same dataset.

In order to tackle overfitting, we added regularization techniques. Batch normalization and dropout were employed throughout the model to improve generalization and to provide a way to control the complexity of the model preventing it from fitting the training data too closely. The results from Table 4 show a better generalization in unseen data with a validation accuracy of 86% compared to the 76% of the complex model. Also, there is a significant drop in validation loss from 1.11 to 0.46 compares to the loss increase of the complex model. Moreover, Figure 4 shows the relationship between the evaluation metrics in both models, and the complex model's validation curves evidently have the desired result.

Hence, by using the simplified model, we have demonstrated how regularization techniques can enhance the model's ability to generalize and improve performance on unseen data.

4. Interpretability

Using our custom CNN model from the previous section (simplified model from 3.4) alongside with the pre-trained model VGG16 [(6)], we are going to visualize the filters and activations

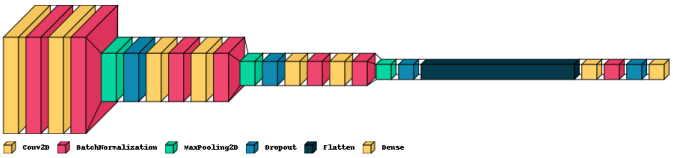

Fig. 6. CNN architeecture of the Simplified model.

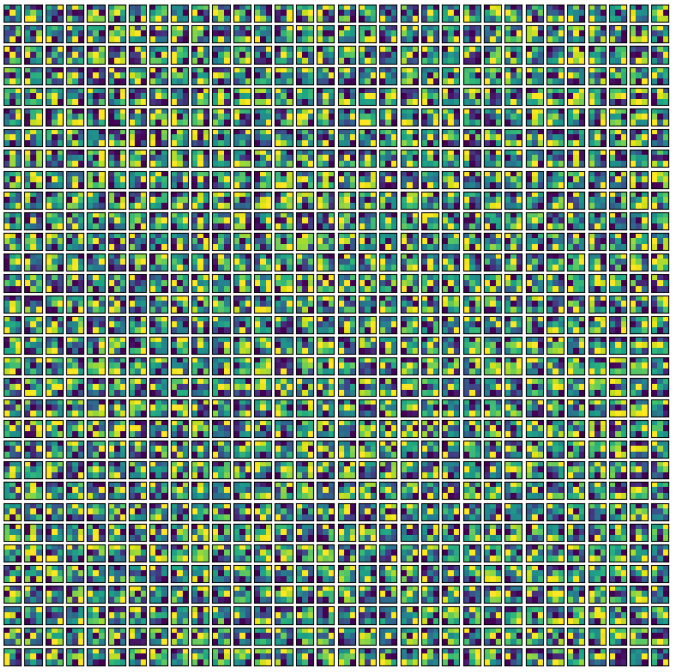

Fig. 7. Filters Visualisation in Custom model.

in the CIFAR-10 dataset and interpret the results. After that, we will get the top k-samples per unit and show the t-SNE plot on each of the dense layers.

4.1. Filters and Activations

In order to visualise the filters, we created a function that takes a convolutional layer as an input, extracts the weights from that layer, and then plots the filters for each channel. A new figure is created with a size of 10×10 inches and two nested loops iterate over the filters *(i)* and input channels *(j)*. Finally, each subplot in the grid represents an individual filter for a specific channel. Figure 7 shows the resulting grid and the color variation in it indicates the weight of a particular feature in the filter. In this particular colormap (*viridis*), dark purple indicate areas where the filter has a weaker response or is not activated strongly, while bright yellow regions indicate a stronger response. This means that the dark regions correspond to parts of the input where the filter does not detect significant features, while bright regions suggest the presence of meaningful patterns or structures. Regarding the VGG16 model, we include the first sample image and its filters in Figure 28. They are way less in number than our custom model, a fact that affects how well this model adapts to the new dataset.

Continuing with the next visualisation, we created another function and used it to visualize the activations of a chosen layer

for multiple sample images from the test dataset. The function plots the original image in the first subplot and the activations for each filter in the subsequent subplots. The number of subplots are calculated based on the square root of the total number of filters in the chosen layer plus one. For simplicity, the first five sample images are chosen, the results of which can be found in the Appendix section 7. What we notice in these results is that the outputs visually display how different filters in the chosen layer respond to the input image, helping us understand what features the neural network is capturing at that specific layer. For example, in the the first in the fourth row of each sample image's grid clearly captures the lighter hues of an image when compared to the original. We also notice filters that capture certain colours, such as the penultimate figure that captures red, and others that capture corners and linear patterns like the third row in each grid.

In order to have a fair comparison between our custom model above and the VGG16 model, we chose to show the activations of the layer 1 instead of layer 0. This decision was based on the fact that layer 0 had an output grid of 2×2 dimension representing low-level features, while our custom model's grid was 6. Choosing layer 1 meant comparing the custom output with a 8×9 grid, which captures more complex features in the images. Due to space limitations the five output grids of the VGG16 model can be found in the Appendix starting from Figures 29 to 33. These activations seem to mainly focus on the textures and shapes, as well as colours. For example, in the Cat figure we notice the yellow parts that capture the colours at the top and bottom of the image. The second activation from the fifth row perfectly captures the shape of the cat. The sample image from Figure 32 is an airplane that contains text at the top of the image and we can see some filters (like the seventh in the penultimate row) capture it.

4.2. Top k-samples per unit

Next, we had to determine which k input samples lead to the highest activations of that particular unit. This approach helps understand what specific feature each unit is responding to. For example, in a convolutional layer, one unit might be most activated by edges in a certain orientation, while another might respond strongly to a specific color. In order to implement that, we created a dictionary where keys are unit indices, and values are lists of indices corresponding to the top-k images for each unit. A grid of subplots based on the number of units and the specified value of k is formed. It then iterates over the units and their top-k indices, plotting the corresponding images in the subplots. The resulting grid of images is displayed in Figure 27. It is interesting to note that the frog image is displayed the most times along the grid, indicating that the image is associated with multiple units, each capturing different aspects or features of the image. It apparently contains features or patterns that are relevant to different aspects of the network's learning. Regarding the VGG16 model, the output of the top k-images consists of 64 rows, the first five of which can be seen in Figure 34. Further down the rows there is a similar pattern to the output of the custom model, which is various rows consisting of the same from image. The repeated appearance of the same frog image could indicate that both models have learned to activate strongly on specific patterns that are indicative of the "frog" class. Another possible reason is that the models might have overfit to certain distinctive features of a frog in the dataset.

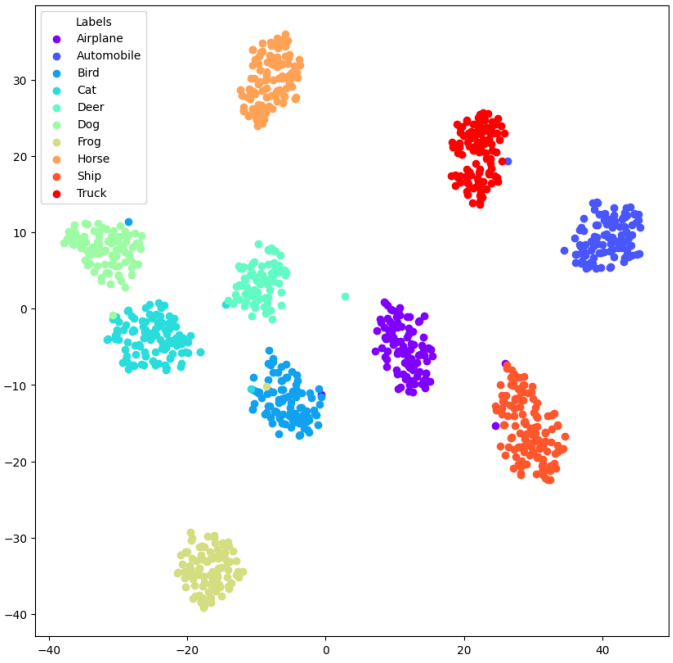

Fig. 8. t-SNE plot for *dense* layer in Custom model.

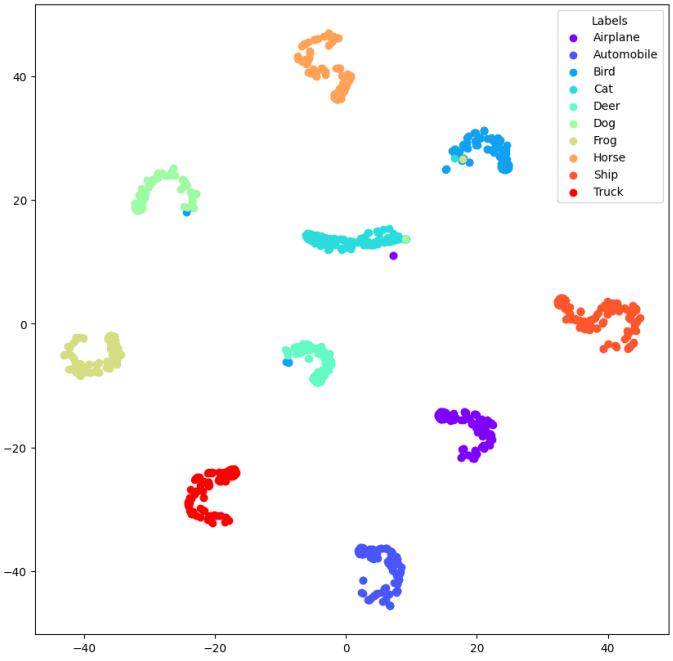

Fig. 9. t-SNE plot for *dense_1* layer in Custom model.

4.3. t-SNE plots in Dense layers

T-Distributed Stochastic Neighbor Embedding (t-SNE) (7) is a dimensionality reduction technique commonly used for visualizing high-dimensional data in a lower-dimensional space, typically two or three dimensions. t-SNE is particularly useful for revealing the inherent structure or patterns in complex datasets, and is often used to visualize the activations of layers. This can help in understanding what features the network is learning and how different layers are responding to different input data.

Since we were met with RAM limitations, we created a subset of the training data in order complete this task. There were two dense layers in our network so we created one plot for each layer (Figures 8 and 9). The first dense layer already does a good job in separating the different features placing them in round clusters, and the missclassifications are almost one for each class. In the second dense layer we can see an improvement in the missclassified images and differently shaped clusters. The smaller and more intricate shapes may indicate that the second dense layer is capturing more nuanced or complex relationships between features. We can suggest that these layers are learning different types of representations or relationships within the data and that the network is effectively learning to distinguish them.

Figures 35 and 36 show the t-SNE plots for the VGG16 models for the two dense layers. The first one fails to distinguish among the different classes, as the points are scattered all over the plot area with a main cluster in the middle containing all types of images mixed together. As we go to the second layer, the classification is not any much better, as now even though there are two different clusters in the plot, the data points are still mixed and the colours are not separate. The custom model clearly does a better job at visualising the data in a lower-dimensional space, as VGG16, being a pre-trained model on a diverse dataset (ImageNet), might not be optimized for the specific features of this dataset.

5. Transfer Learning

For the transfer learning part of the assignment, we followed the statements provided. We used the Terrassa-900 dataset, a dataset of still images of buildings from the city of Terrassa.

We loaded the data with the splits for train, test and validation provided in the webpage.

The preprocessing we started performing was normalizing each image by 255, and add some data augmentation in the train set:

- Rotation Range: 20 degrees
- Width Shift Range: 0.2
- Height Shift Range: 0.2
- Shear Range: 0.2
- Zoom Range: 0.2
- Horizontal Flip: Enabled
- Fill Mode: Nearest

Normalization, which is the process of scaling and centering numerical data so that they have a consistent scale and was done by converting every image into a NumPy array and then normalizing the pixel values by dividing them by 255. This practice will make sure that the pixel values have now a range of [0,1] and will help prevent numerical instability during training, especially when using certain activation functions. Also, the gradient updates will be consistent across different values and it is not uncommon to also help with the speed of the convergence.

Data augmentation, another famous preprocessing technique, artificially increases the size of the training set by applying various transformations to the existing data. In this way, diverse variations of the original data are created providing the model with a more extensive and varied set of examples to learn from.

The optimizer we used was Adam (short for Adaptive Moment Estimation), as it is found in the literature that perform well across a wide range of tasks and input types. Adam optimizer combines ideas from two other optimization methods — momentum and RMSprop — and introduces adaptive learning rates for each parameter (3).

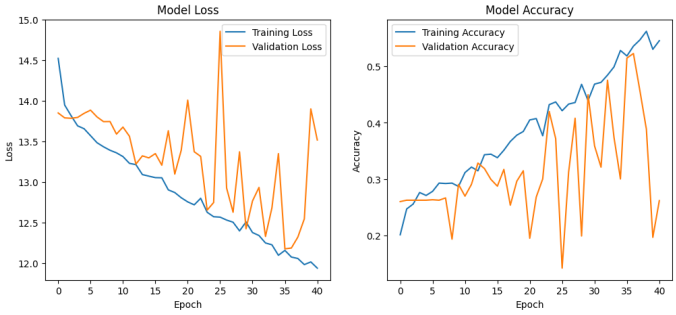

Fig. 10. Results of the tweaked model for Terrassa-900.

The batch size we used was generally 256, and the loss was Sparse Cross Entropy with a learning rate of 0.01, but other combinations were also used with similar results for the following sections.

5.1. CNN on Terrassa 900 dataset

The first approach was to create a CNN with single convolutional layers (Figure 11) with some regularization. This resulted in underfitting, with a mean accuracy in train set of 0.2639 and 0.2646 in validation.

More complex models using sequential convolutional blocks presented similar results.

After these results, we tried with another preprocessing step, featurewise centering the images and resizing them to (150,150), similar results arised, but were a little bit better (Figure 10). The goal of feature-wise centering is to subtract the mean of each feature across the entire dataset independently. This process results in each feature having a mean of zero.

5.2. Feature Extractor

Then, we tried a CNN as a Feature Extractor on the Terrasa-900 dataset. For this, we trained a network on CIFAR-10. The architecture of the network had two convolutional blocks, one formed by two convolutional layers of 32 units with a kernel of size (3,3), and the other of 64 units. After each block, a MaxPooling Layer and a Dropout for regularization was added. Finally, a Dense layer of 512 units with a Dropout of 0.5

After the long time it took to train the first models, we added a EarlyStopping mechanism with a patience of 5, that stops after the number of epochs with no improvement after which training will be stopped using the validation loss.

The results in Figure 13 show a train and test accuracy of around 80%.

After training the feature extractor, we proceeded to train a Support Vector Machine (SVM) classifier using cross-validation. The SVM was configured with the following parameter grid:

$$\text{param\_grid} = \{ \quad 'C' : [0.1, 1, 10], \quad 'gamma' : [0.001, 0.01, 0.1, 1] \quad \}$$

We utilized the radial basis function (RBF) kernel and set the random seed to 42 for reproducibility.

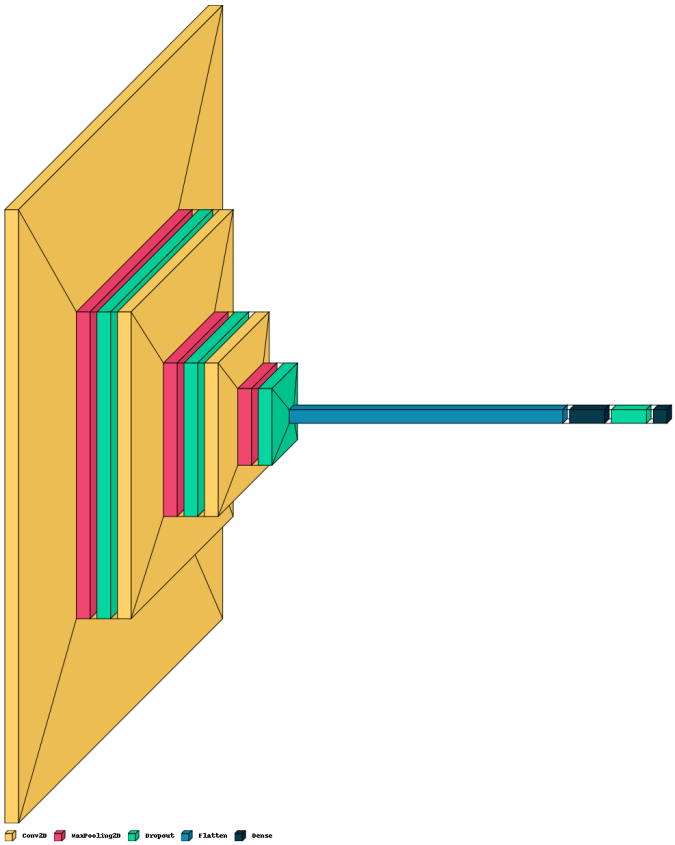

Fig. 11. CNN architecture of the first model for Terrassa-900. Yellow layers are Conv2D, red MaxPooling, green Dropout, blue Flatten and dark blue Dense

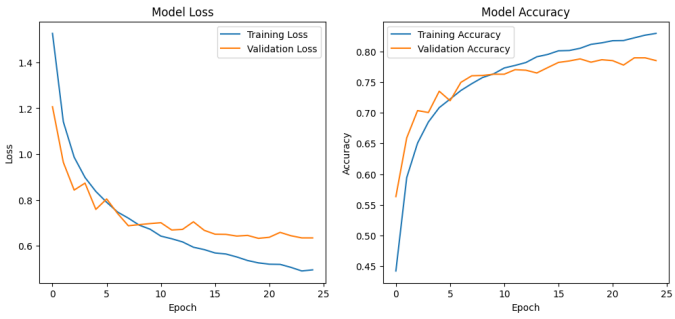

Fig. 12. Losses and accuracy for the CNN trained in CIFAR-10.

Upon grid search, the best-performing combination of hyperparameters was found to be 'C': 10, 'gamma': 0.01, resulting in an accuracy of 0.46 on the validation set.

On the Confusion Matrix in Figure 13 we can see a correct classification in several categories, but a lot of false positives in the class number 3, Societat General d'Electricitat, probably because this is the most common class in the dataset.

5.3. Fine Tuning

After the trained CNN over CIFAR, we used VGG16 with ImageNet weights, removing the last layer and freezing all the other layers but the last two. We added two simple layers with 1024 dense units and 512, with Dropouts of 0.2 and ReLu activation

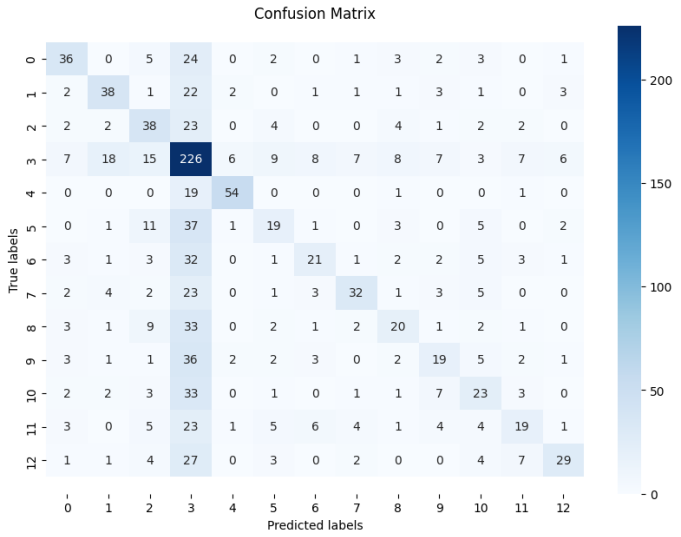

Fig. 13. Confusion matrix of the SVM using the Feature Extractor.

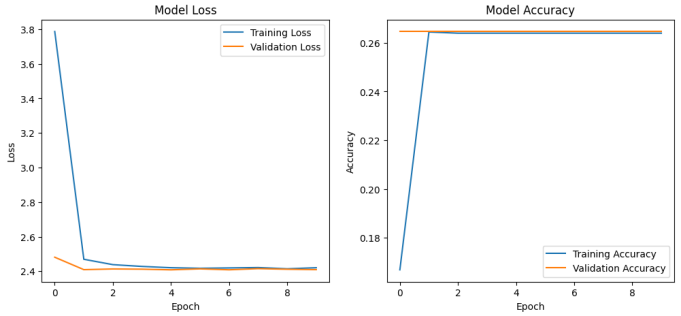

Fig. 14. Losses and accuracy for the Network with VGG16 pretrained in Terrassa-900.

functions. The results were shockingly bad, as we can see in Figure 14, where the train and test accuracy seem to converge at 0.26

5.4. More Fine Tuning

For this last part, we used the UoS-Building Dataset (1) to fine-tuning a model with important building features for our classification task in Terrassa-900. This dataset contains building images from the University of Stalford:

1. Chapman: 74 images
2. Clifford Whitworth Library: 60 images
3. Cockcroft: 67 images
4. Maxwell: 80 images
5. Media City Campus: 92 images
6. New Adelphi: 93 images
7. New Science, Engineering & Environment: 78 images
8. Newton: 92 images
9. Sports Centre: 55 images
10. University House: 60 images

For the architecture, the VGG16 model, loaded with ImageNet weights, serves as the base, with its last four layers unfrozen for fine-tuning. Custom layers are added, including flattening, two dense layers (1024 and 512 units respectively) with

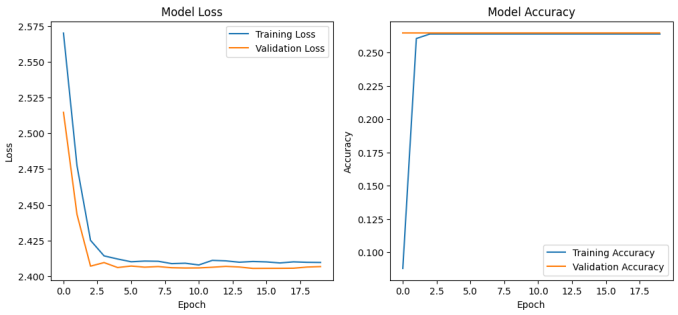

Fig. 15. Losses and accuracy for the Network with VGG16 pretrained in UoC-Dataset

ReLU activation and dropout of 0.2, and a final dense layer with softmax activation.

In Figure 15 we can see the lossess and accuracies in the train and validation sets. We can see a clear underfitting, which seems really strange by the fact that we are using a pretrained VGG16 in Imagenet and also in our buildings dataset.

5.5. Discussion

The results seem bad, as we did not expect that using pre-trained networks such as VGG16, with an architecture capable of tackle our problem, we would have underfitting. We attribute this problem to some problem with the tensorflow library, which may be doing something behind the scenes, or some problem with the data input. We tried changing hyperparameters like the loss, the regularization, the architecture, the number of epochs, converting the images to grayscale, reducing the size of the images, etc., but nothing seemed to help.

6. Satellite Image Classification

In the last part of our analysis, we are taking an independent dataset and, after choosing an appropriate resampling method, create a CNN model and trying to get a good score. The chosen dataset is comprised of Remote Sensing images gathered by a satellite. As NASA defines, remote sensing is the acquiring of information from a distance. NASA observes Earth and other planetary bodies via remote sensors on satellites and aircraft that detect and record reflected or emitted energy. Remote sensors, which provide a global perspective and a wealth of data about Earth systems, enable data-informed decision making based on the current and future state of our planet. Deep neural network (DNN) can be used on these big data datasets and offer tranferable machine-learning technologies. Even though this practice is considered to be in its infancy as there are many challenges to be faced as well as potential advancements in the application of DNNs (4), many diverse and representative benchmark datasets for remote sensing have been developed.

The dataset we will be using onwards is called Satellite Remote Sensing Image -RSI-CB256, which is one of the six categories of the remote sensing image classification benchmark (RSI-CB) and consists of six categories (construction, transportation, agriculture, water, woodland, and other land) with 35 sub-classes. The hierarchical grading system is based on the Chinese land-use classification standard, aiming for diversity and comprehensiveness. The authors of the respective paper (5) use crowdsourced geographic data from Open Street Map (OSM) to annotate remote sensing images. The data is aligned with images from Google Earth and Bing Maps, creating a benchmark with diverse ground objects.

6.1. Data Exploration

Our dataset includes 5631 images from the Earth's surface split into four categories:

(a) cloudy - 1500 images
(b) desert - 1131 images
(c) green area - 1500 images
(d) water - 1500 images

As we can see, the classes are not balanced, so some actions will have to be taken to deal with it during preprocessing. Figure 16 displays a sample of the images of each class, to show the quality and the contents of the images. They are quite zoomed-in, so a basic feature of them is the colour as most of them have faint distinct textures and shapes in them. It is interesting to see how the top middle image looks like a water sample at first sight. That might be due to the lack of sun when the image was sampled or the long distance between the satellite and the forest.

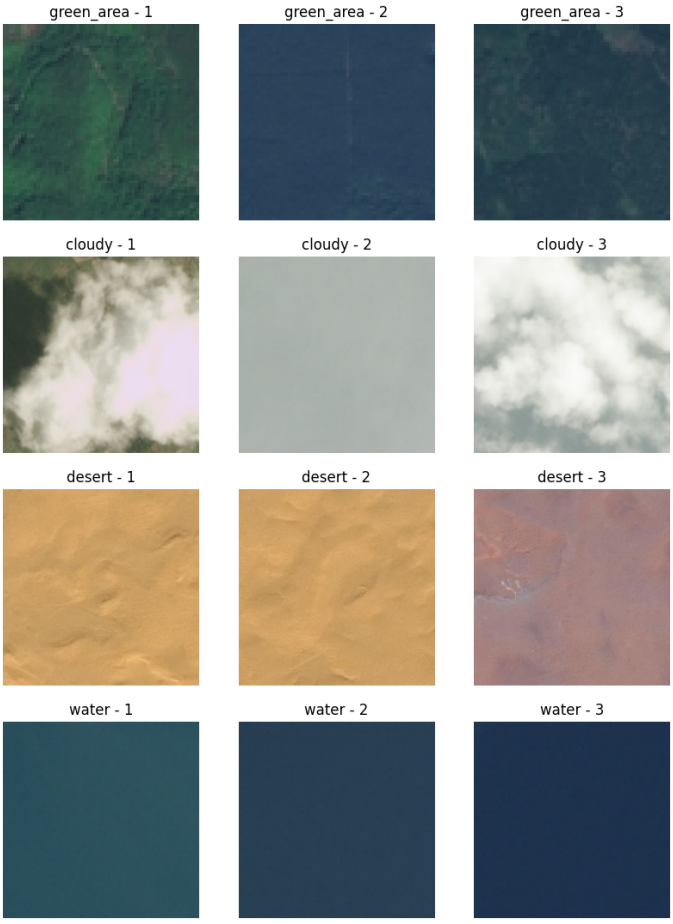

Fig. 16. Sample images for each class.

The images have two different dimensions: Green area and water images are 64×64, while cloudy and desert are 256×256. This difference indicates that a resizing should be implemented before feeding the images to the neural network.

6.2. Preprocessing

For the part of preprocessing we started by resizing the images. As seen in the previous section, the images across the different classes did not have the same size. So, in order to ensure that they can be efficiently processed in each batch, we changed their dimension to 224×224 pixels. Also, as we will be using the VGG16 pretrained model later in the analysis, resizing guarantees compatibility with the model.

In this dataset, the desert class only has 1131 images compared to the 1500 images of the other categories, so we performed SMOTE to create synthetic data for this class in the training set (2). SMOTE, which stands for Synthetic Minority Over-sampling Technique, is a technique used to address the issue of class imbalance in machine learning datasets. Class imbalance occurs when the distribution of instances across different classes is not equal, as in our case. This imbalance can lead to biased models that perform poorly on the minority class, as they might not learn sufficient information from the underrepresented class. For each minority class instance, SMOTE selects k-nearest neighbors. It then generates synthetic instances by interpolating between the original instance and its neighbors. The number of synthetic instances generated is determined by a user-defined parameter called the "sampling strategy" or "oversampling ratio." It's important to apply SMOTE only to the training set, ensuring that the validation and test sets reflect the real-world distribution. In extreme cases, metrics like accuracy can be misleading because a model may achieve high accuracy by simply predicting the majority class most of the time. Our case is not considered to be extreme as the difference in 350 images is almost trivial. The whole analysis could have been carried out without implementing SMOTE and the results would apparently not differ much, but, as we had never used it before, we wanted to face the challenges in its use and gain some experience. While SMOTE is a powerful technique, it may not be suitable for all scenarios; the opinions are controversial. For instance, in cases where the minority class is already well-represented or when synthetic examples could introduce noise, other techniques might be more appropriate.

Finally, we performed data augmentation, a method also mentioned in Section 5.

6.3. Whitening

We first went with a different approach, we used PCA to whiten the images. The goal is to decorrelate the features (pixels) and make them more independent. This can help in image classification tasks because it simplifies the data by removing unnecessary information, reduces noise, and highlights the most informative features. After checking the Explained Variance plot in Figure 17, by only selecting 3 components we got 97.4% of variance explained.

As PCA reshapes the input data, we could not use a CNN approach. Other techniques like zero-phase component analysis (ZCA) where considered but not implemented due to time constraints. We then applied a simple architecture of Dense layers of 512, 256, 128 units with BatchNormalization and Dropouts of 0.5, 0.4 and 0.3. The results are stuck at a train loss of 0.368, a train accuracy of 0.89; and validation loss of 2.579 and validation accuracy of 0.274.

In this case we can see a clear overfitting pattern, shown in the Confusion Matrices in Figures 18 and 19.

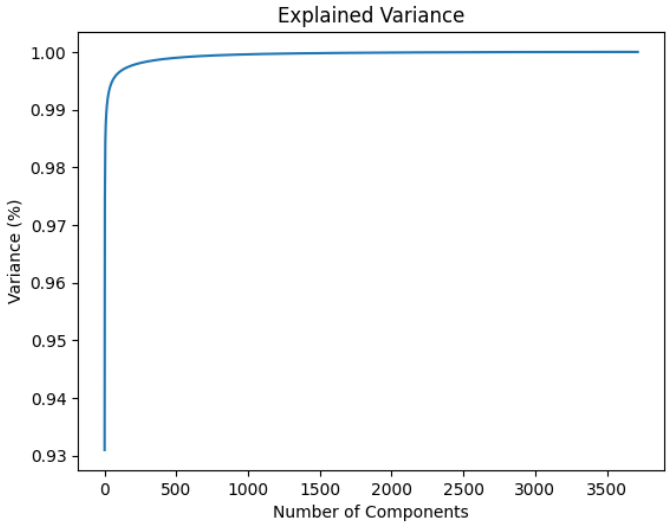

Fig. 17. Explained variance after PCA over RSI-CB256

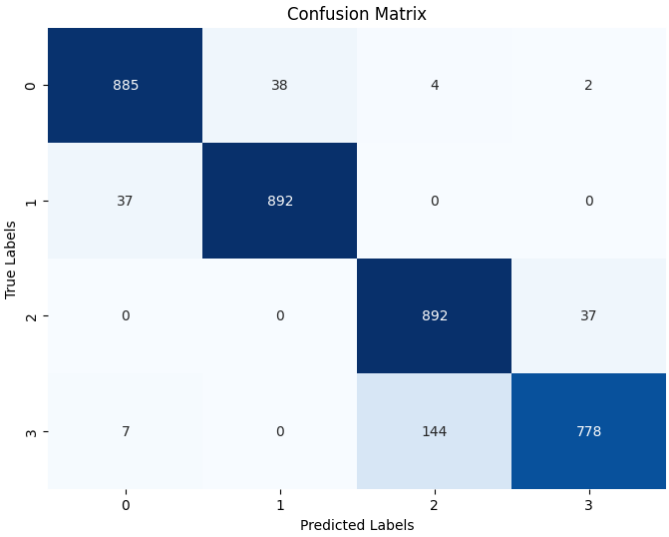

Fig. 18. Confusion matrix on the train set in the NN after PCA over RSI-CB256

6.4. CNN models

For the first CNN model we tried, we used an architecture that consists of several layers. It begins with two convolutional layers with 64 units each, followed by max-pooling and dropout layers to reduce dimensionality and prevent overfitting. This pattern is repeated with two more pairs of convolutional layers, increasing the number of units to 128 and then 256, while maintaining max-pooling and dropout. Additionally, two more convolutional layers with 512 units each are introduced before flattening the output. The fully connected layers include a dense layer with 1024 units, followed by batch normalization and dropout, and another dense layer with 256 units, batch normalization, and dropout. Finally, the model has an output layer with 4 units using softmax activation for classification.

Then, for training we used 100 epochs and a batch size of 64 for computational reasons; and using Adam, Categorical Cross Entropy and an Early Stopping.

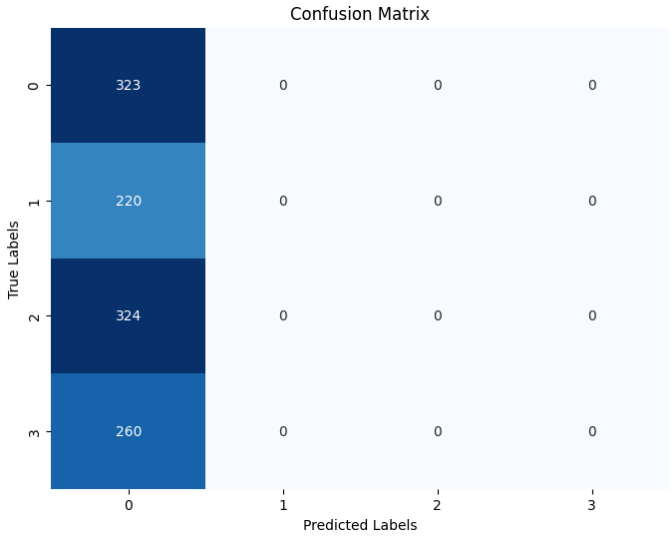

Fig. 19. Confusion matrix on the test set in the NN after PCA over RSI-CB256

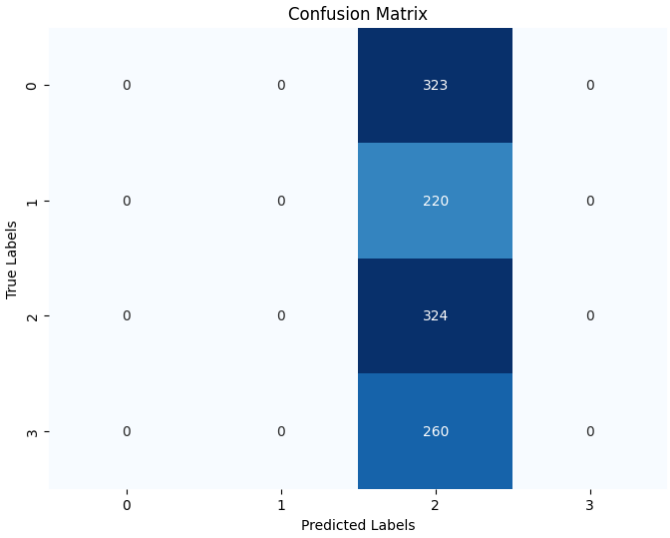

Fig. 21. Confusion Matrix for first CNN model in RSI-CB256

In Figure 20 we see a similar overfitting pattern as before. Figure 21 gives a result where the model is only predicting class 2, similar to the plots shown above, where the PCA model was only predicting class 0.

After these results, we went back and forth with more complex and simpler models:

- A model with only one convolutional block of 3 layers of 64 units, with a (3,3) Kernel and a Dense layer before the output with BatchNormalization and Dropout.
- A model based in MobileNetV2 with an extra layer, only training the last 100 layers with a high Dropout at the end of 0.9

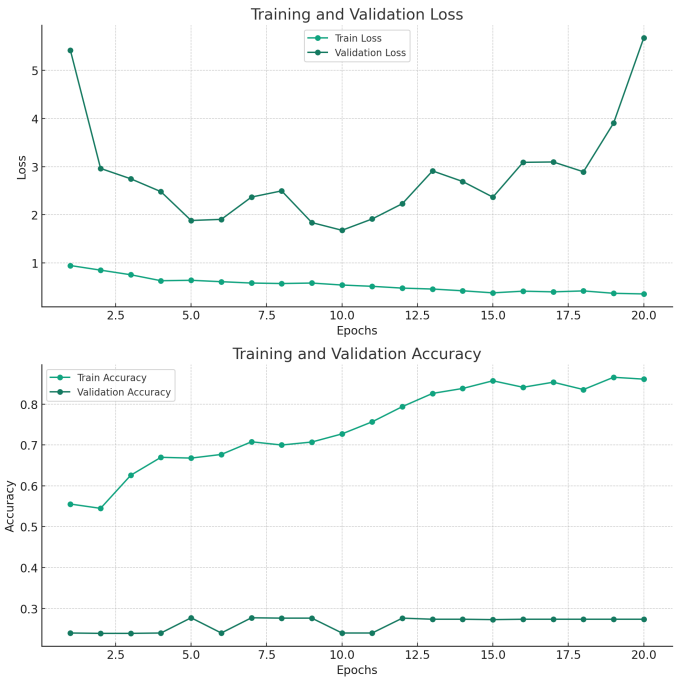

Fig. 20. Validation-Accuracy curves for first CNN model in RSI-CB256

- A model based in ResNet9, written in Keras

With these models, same results emerged, only predicting one class. For the first model, the one with only one convolutional block, there were also some predictions to class 2 (however, it was almost always predicting class 1) but was always wrong.

6.5. Conclusions

Similar results were obtained by other users on Kaggle, so it seems like the dataset that we selected may be difficult for traditional CNN. This is due to several facts, as while CNNs are excellent at capturing local spatial features, they may struggle to capture global context in large satellite images. Understanding the relationships between objects or features across a wide area can be challenging. Also, important features might be sparsely distributed (for instance, detecting specific types of buildings or vegetation). CNNs are designed for dense grid-like data. Satellite images often require specialized preprocessing steps such as atmospheric correction, radiometric calibration, etc. Preprocessing steps out of the scope of the course.

7. Future Work

Our analysis was held under a limited time period and not efficient computational capacity. Resizing the images was taking up the RAM and training the models was time consuming and needed high GPU usage. In a future analysis, these problems can be tackled and more approaches can be tried in order to unravel the true issue of pre-trained models not generalising well or the reason why satellite images as an input make the CNNs struggle and not perform as expected. We believe that other types of preprocessing techniques should be applied as the CNN themselves have proved efficient in a variety of applied problems through different sectors.

References

[1] Ali Alameer and Mazin Al-Mosawy. UoS Buildings Image Dataset for Computer Vision Algorithms. 8 2022.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, June 2002.

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[4] Jiayi Li, Xin Huang, and Jianya Gong. Deep neural network for remote-sensing image interpretation: status and perspectives. *National Science Review*, 6(6):1082–1086, 05 2019.

[5] Yang Long, Gui-Song Xia, Shengyang Li, Wen Yang, Michael Ying Yang, Xiao Xiang Zhu, Liangpei Zhang, and Deren Li. On creating benchmark dataset for aerial image interpretation: Reviews, guidances, and million-aid. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 14:4205–4230, 2021.

[6] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[7] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

Appendix A - Interpretability

Activation Visualisation for Custom Model for the first 5 sample images:

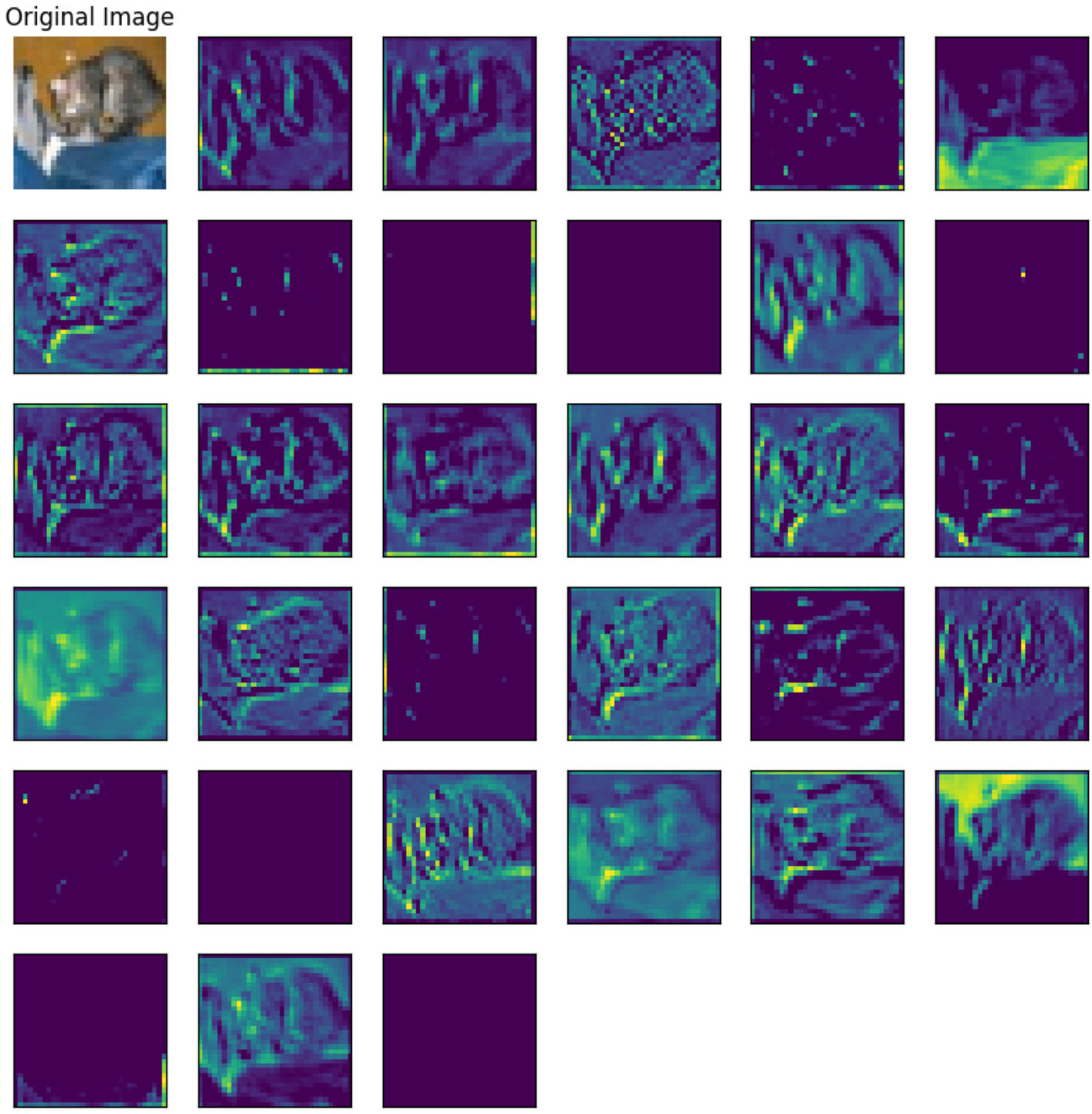

Fig. 22. Custom model Sample Image 1 Activations - Cat.

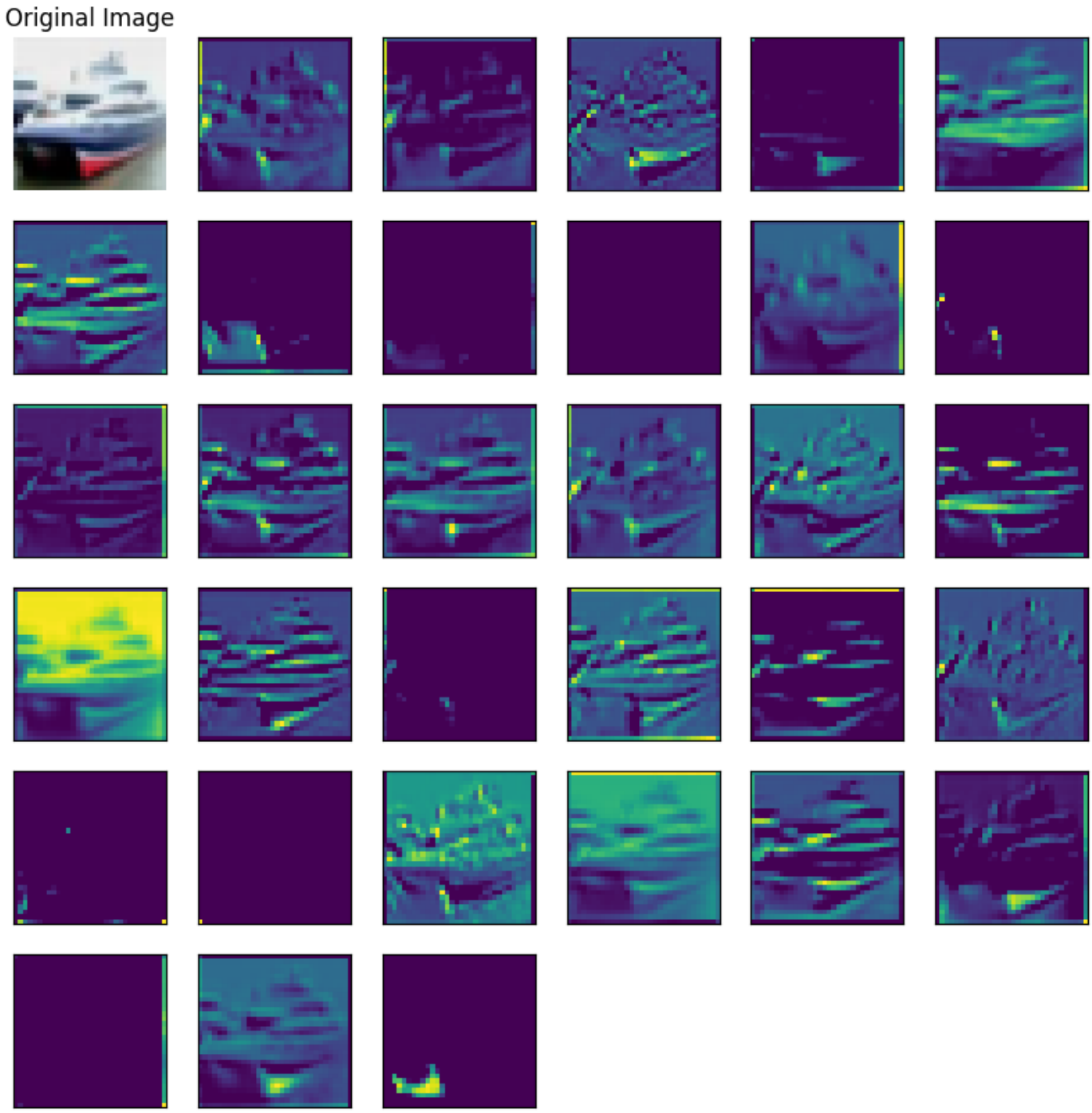

Fig. 23. Custom model Sample Image 2 Activations - Ship.

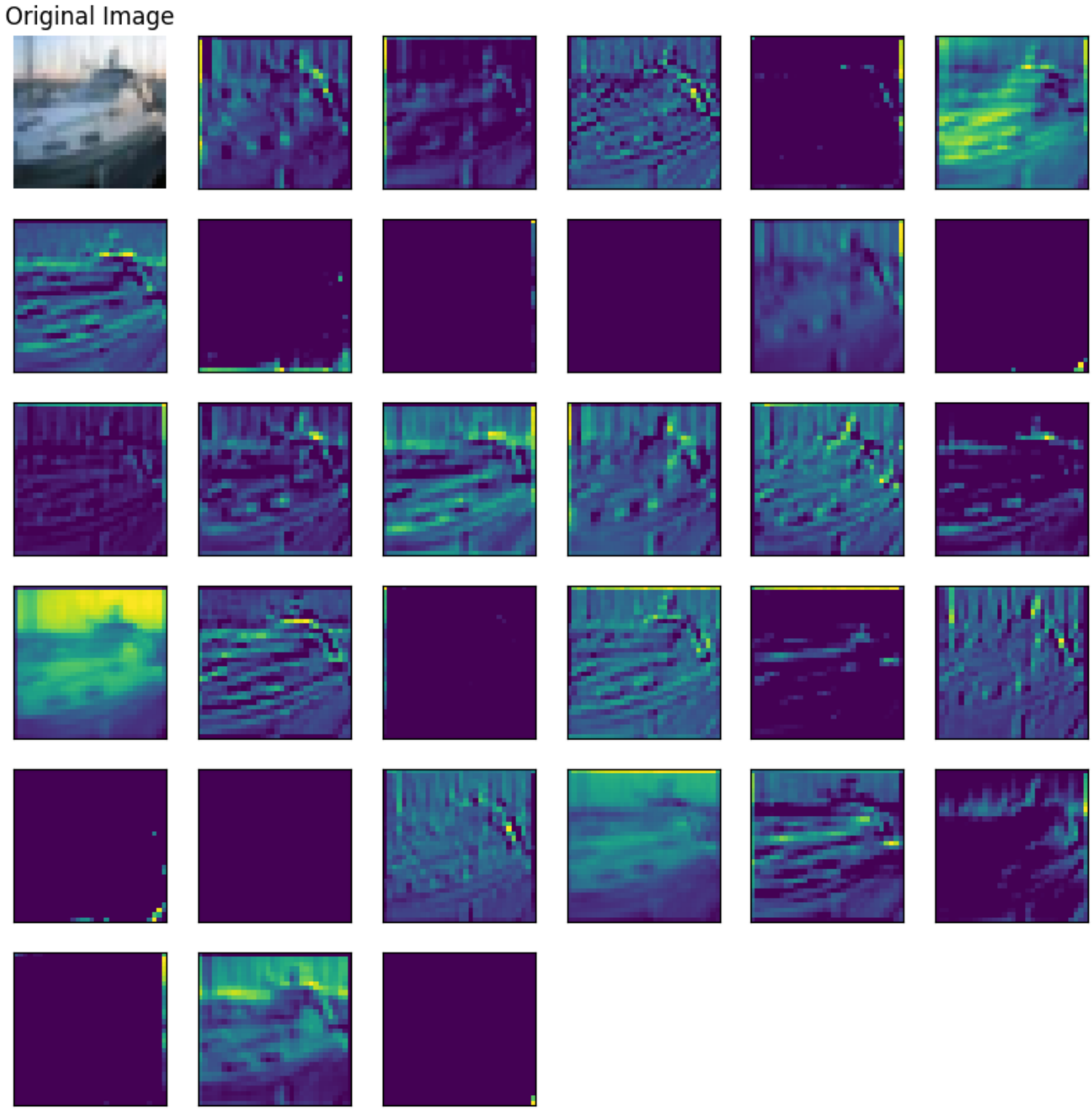

Fig. 24. Custom model Sample Image 3 Activations - Ship.

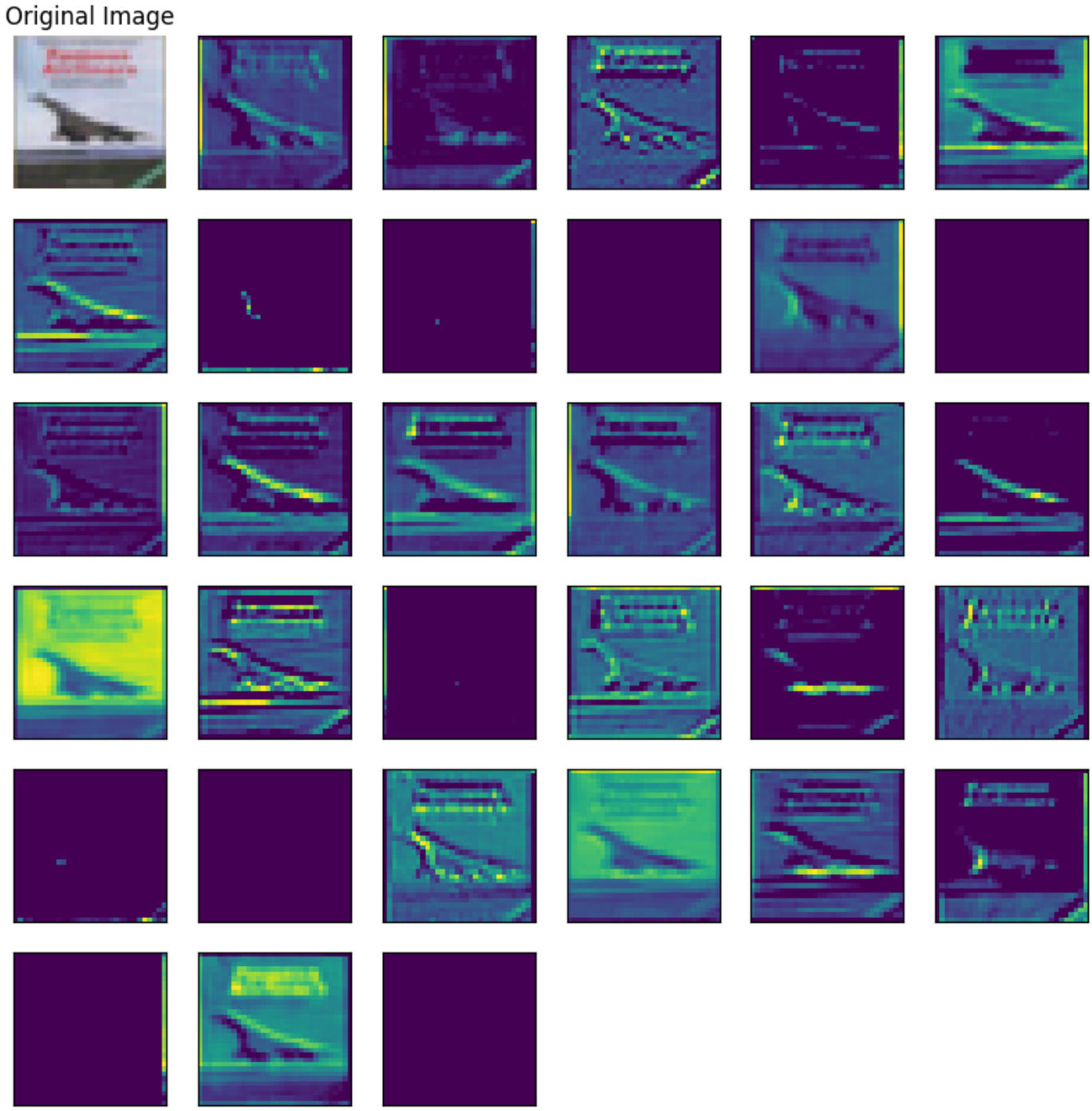

Fig. 25. Custom model Sample Image 4 Activations - Airplane.

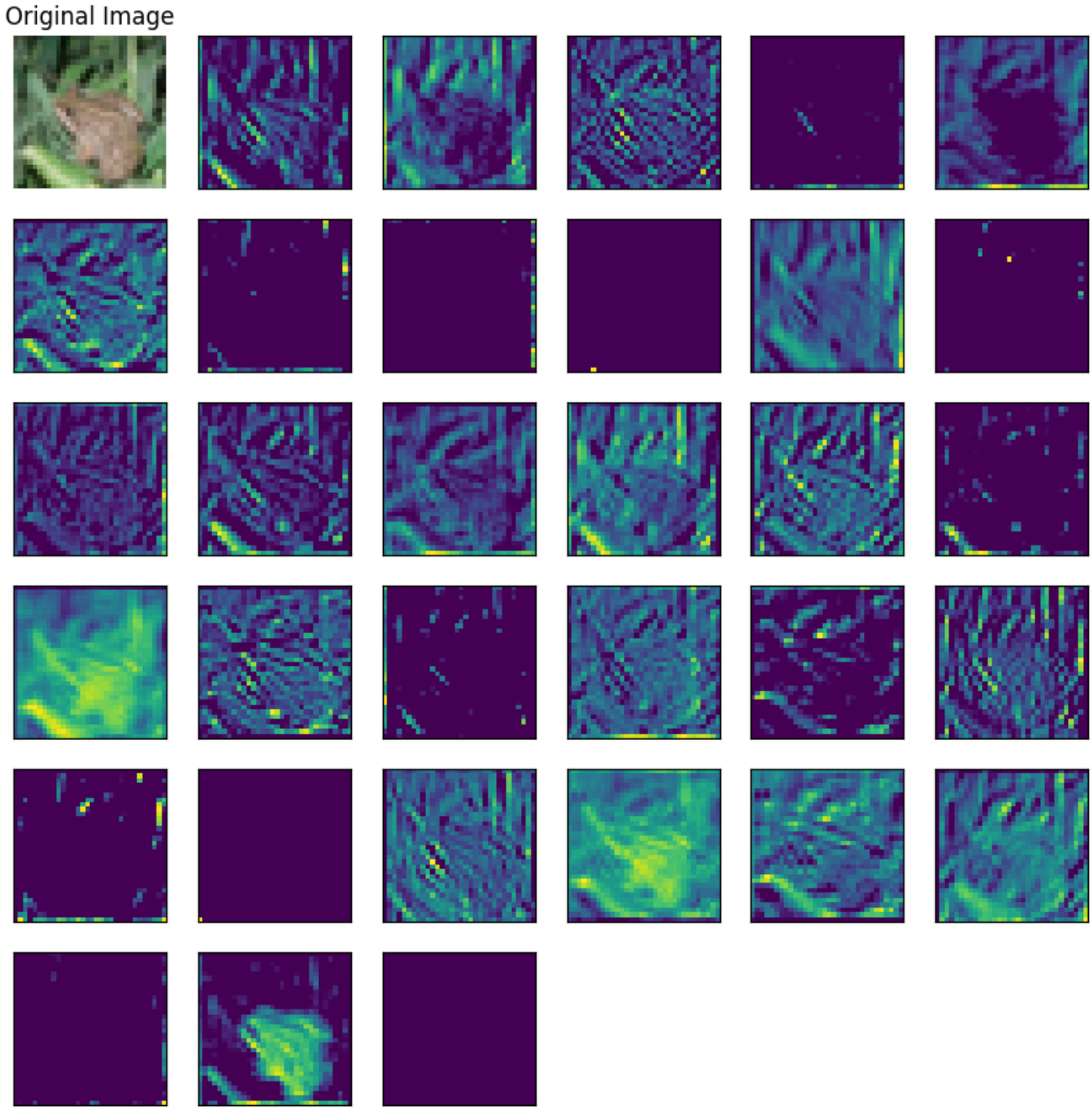

Fig. 26. Custom model Sample Image 5 Activations - Frog.

Top k-images in Custom model:

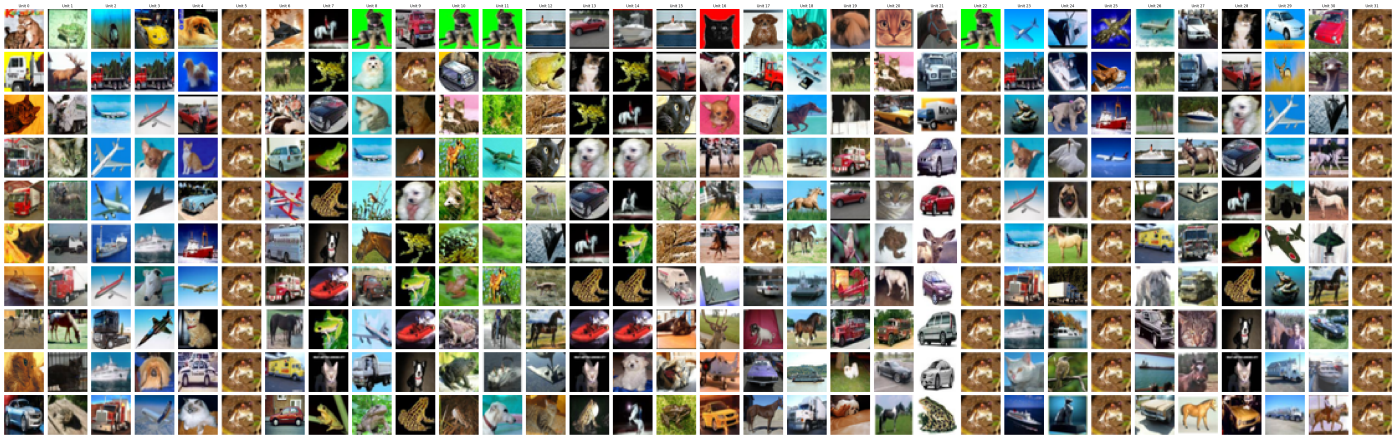

Fig. 27. Top k-images in Custom model.

VGG16 Dataset
Filters Visualisation for VGG16 Model for the fist sample image:

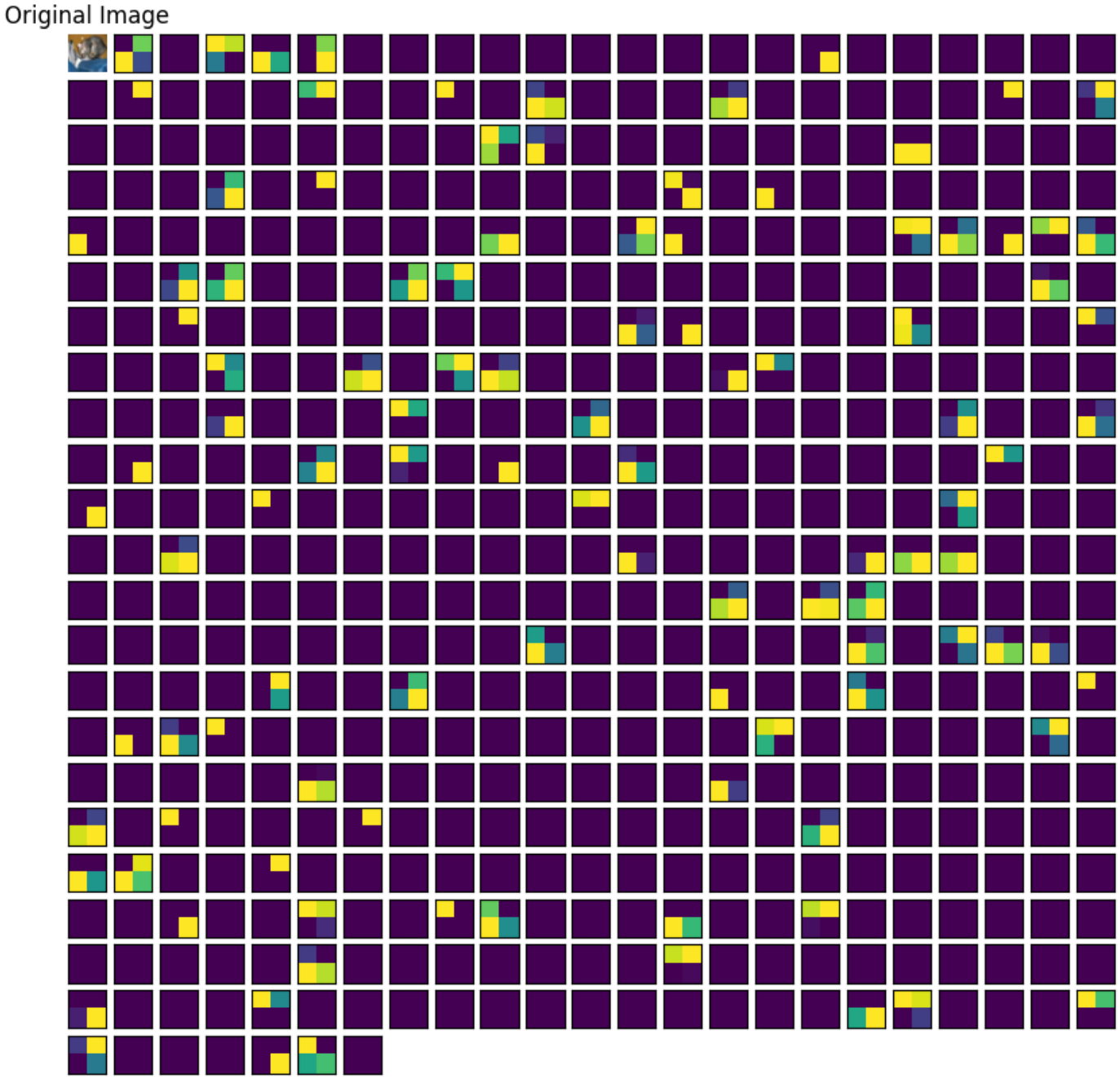

Fig. 28. VGG16 Sample Image 1 Activations - Cat.

Activation Visualisation for VGG16 Model for the first 5 sample images:

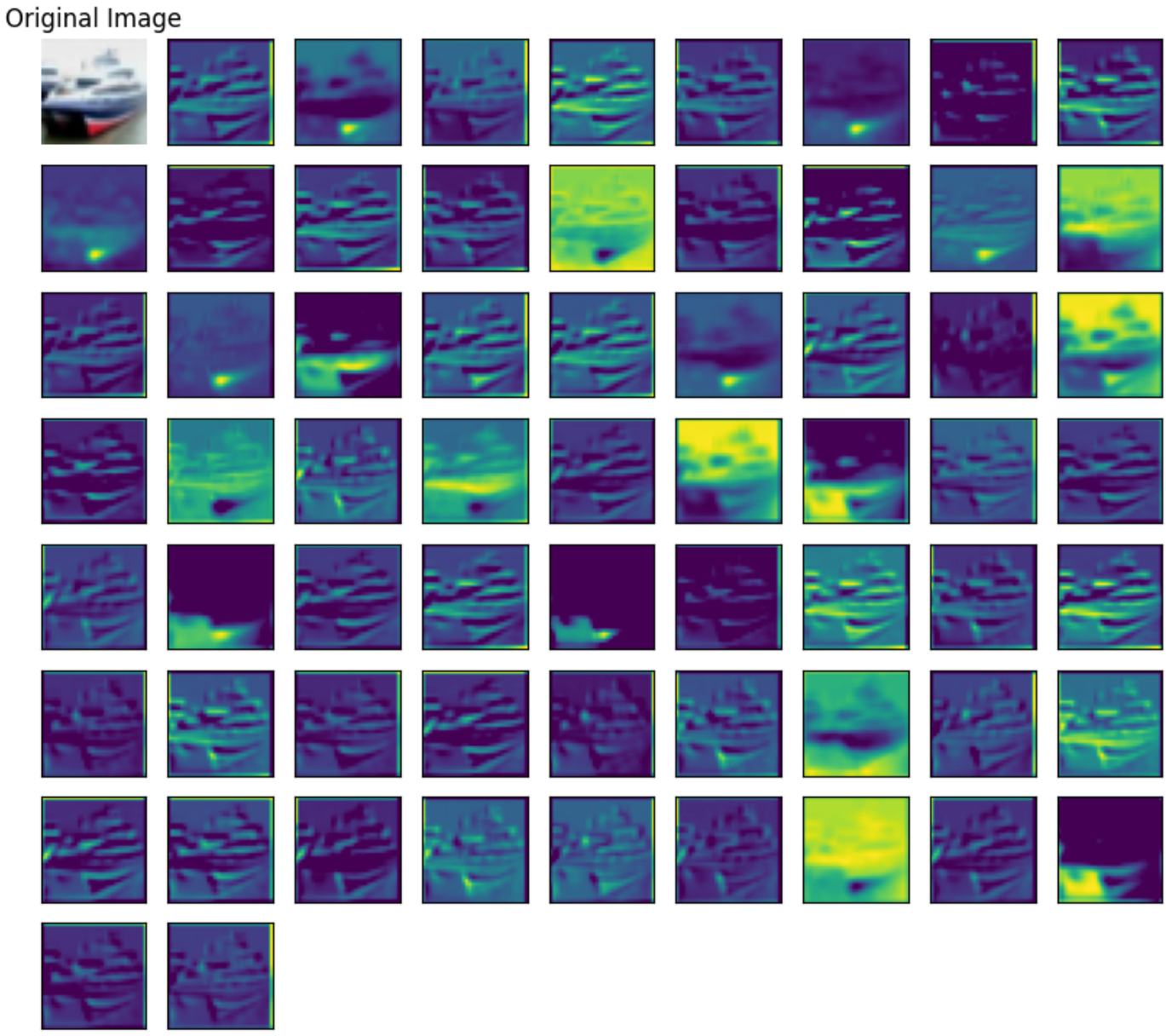

Fig. 30. VGG16 Sample Image 1 Activations - Cat.

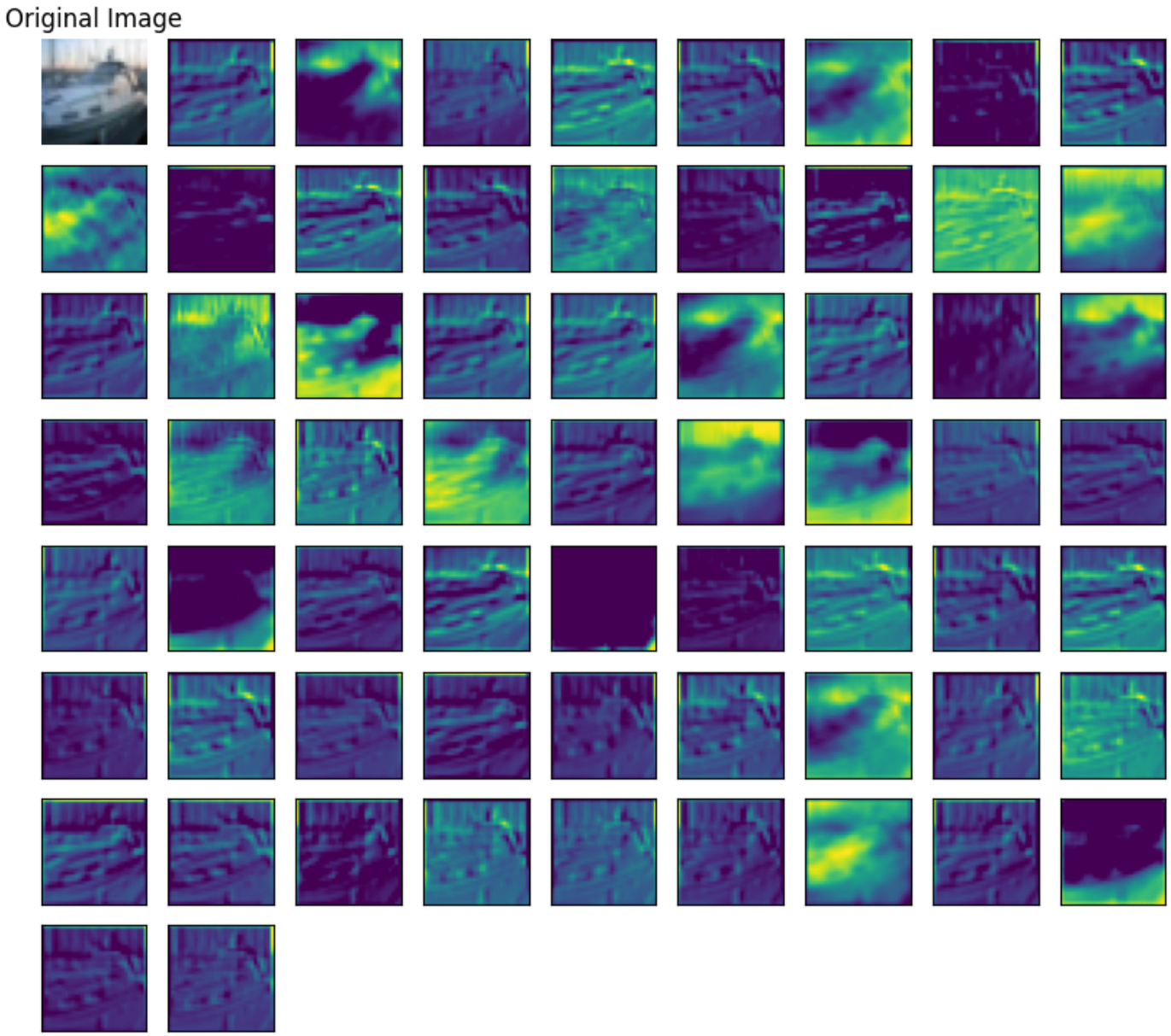

Fig. 31. VGG16 Sample Image 1 Activations - Cat.

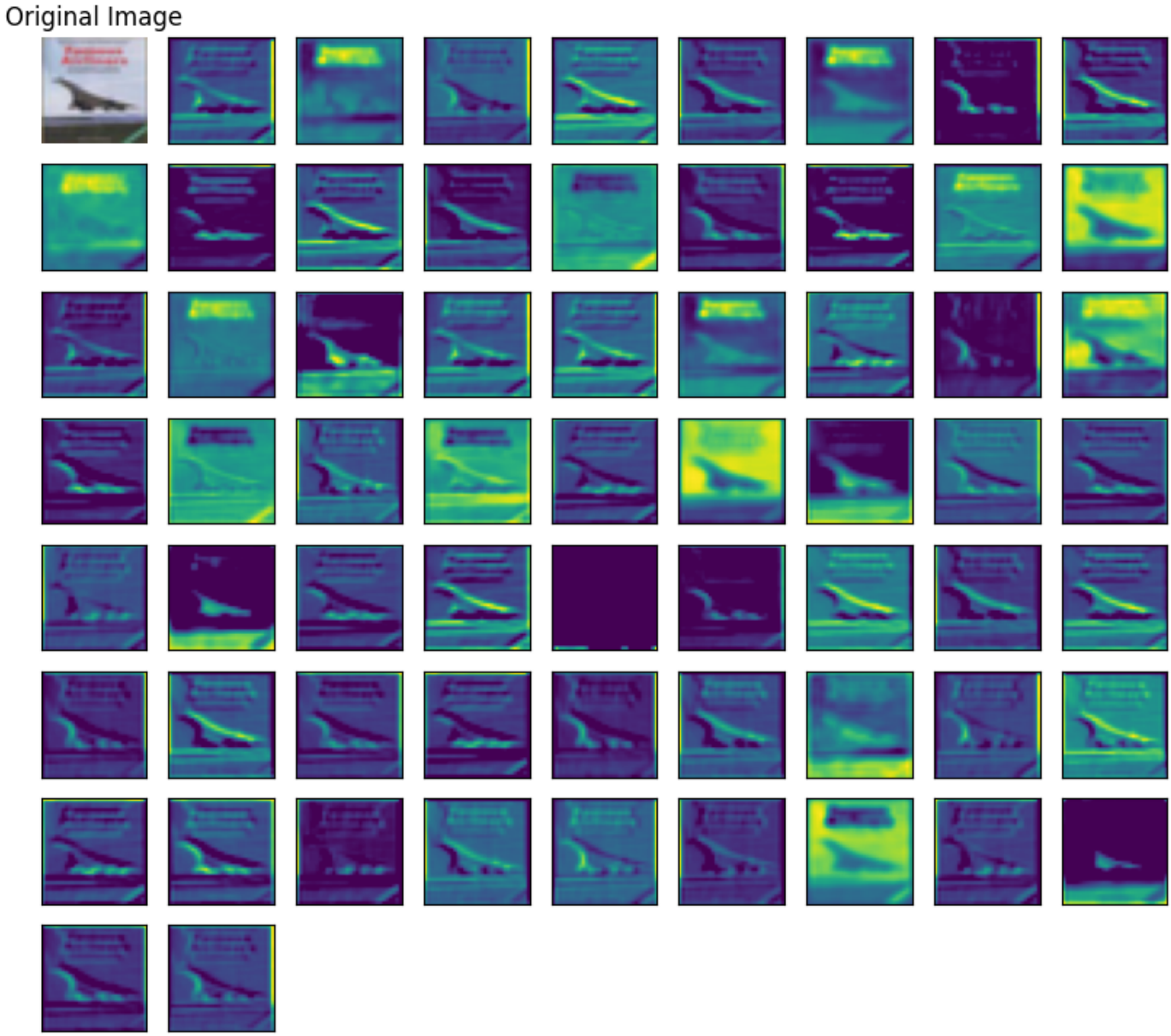

Fig. 32. VGG16 Sample Image 1 Activations - Cat.

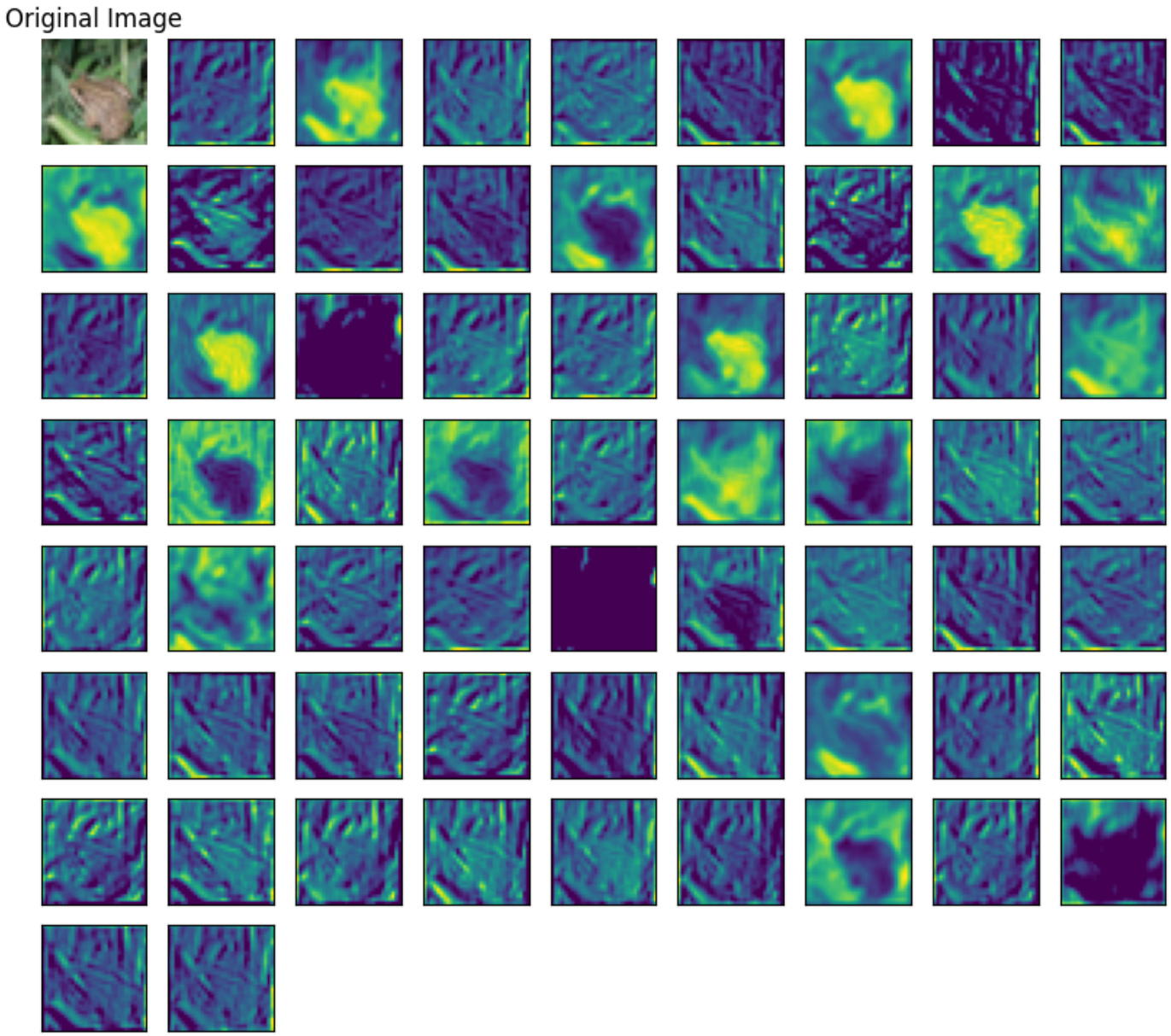

Fig. 33. VGG16 Sample Image 1 Activations - Cat.

First five rows of Top k-images output in VGG16 model:

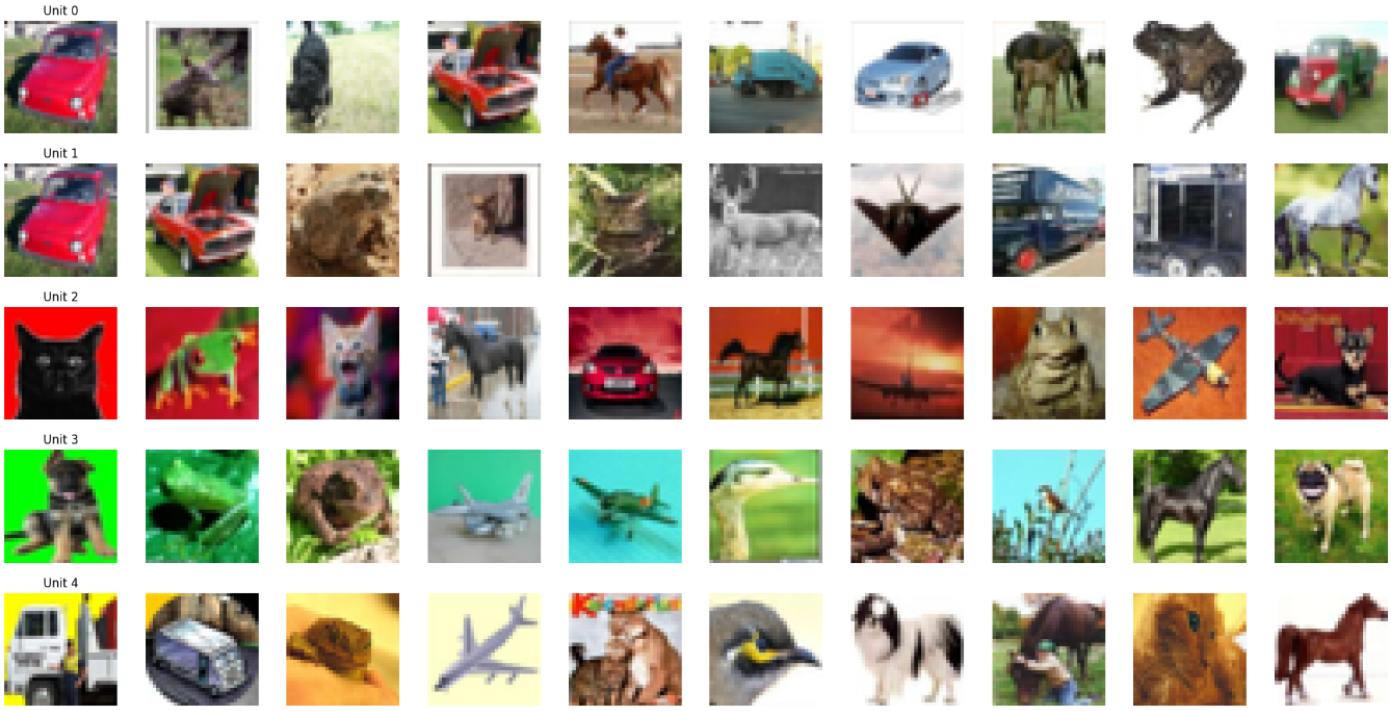

Fig. 34. First five rows of Top k-images output in VGG16 model.

t-SNE plots in Dense layers in VGG16 Model:

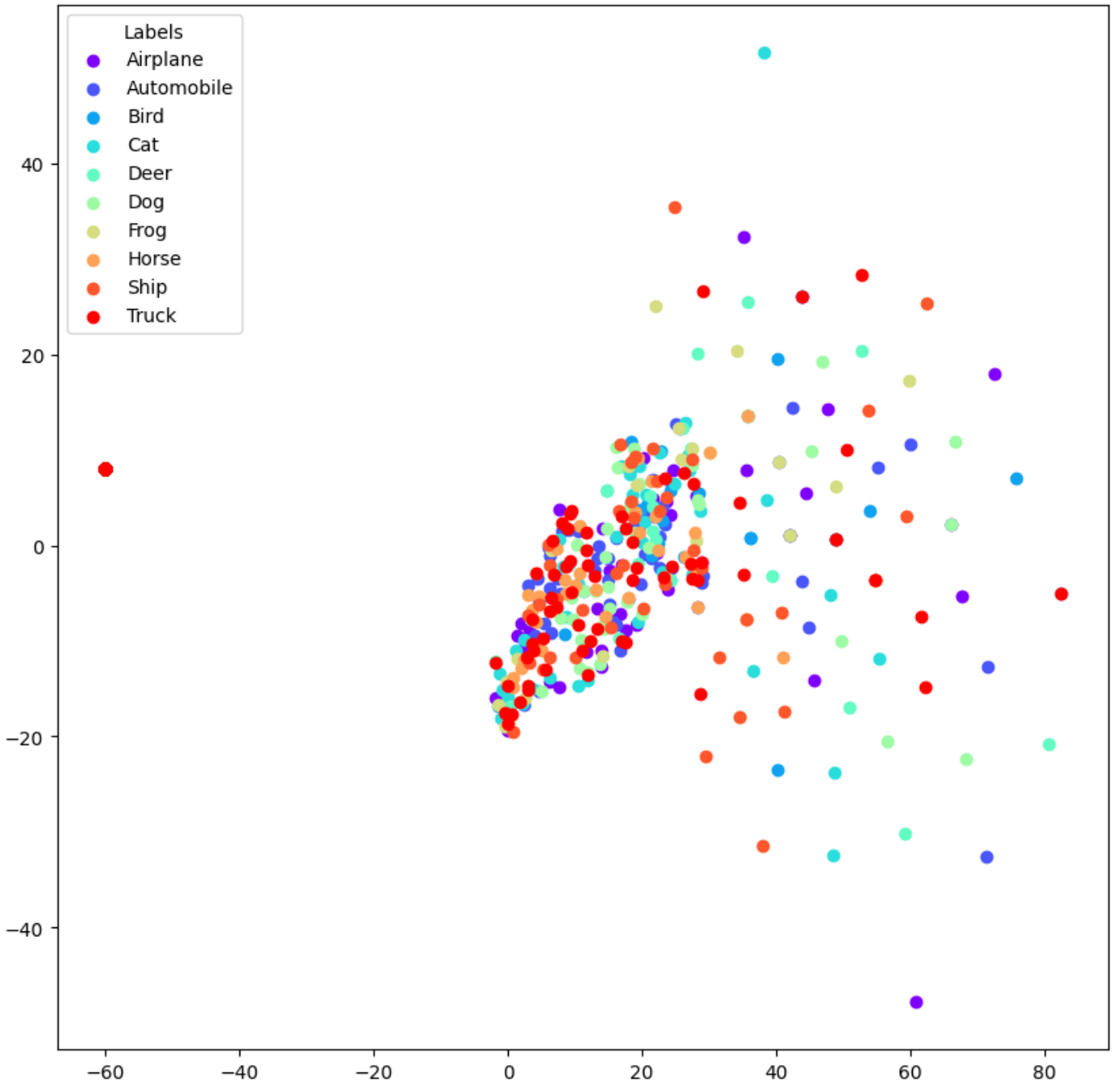

Fig. 35. t-SNE plot for *fc1* layer in VGG16 model.

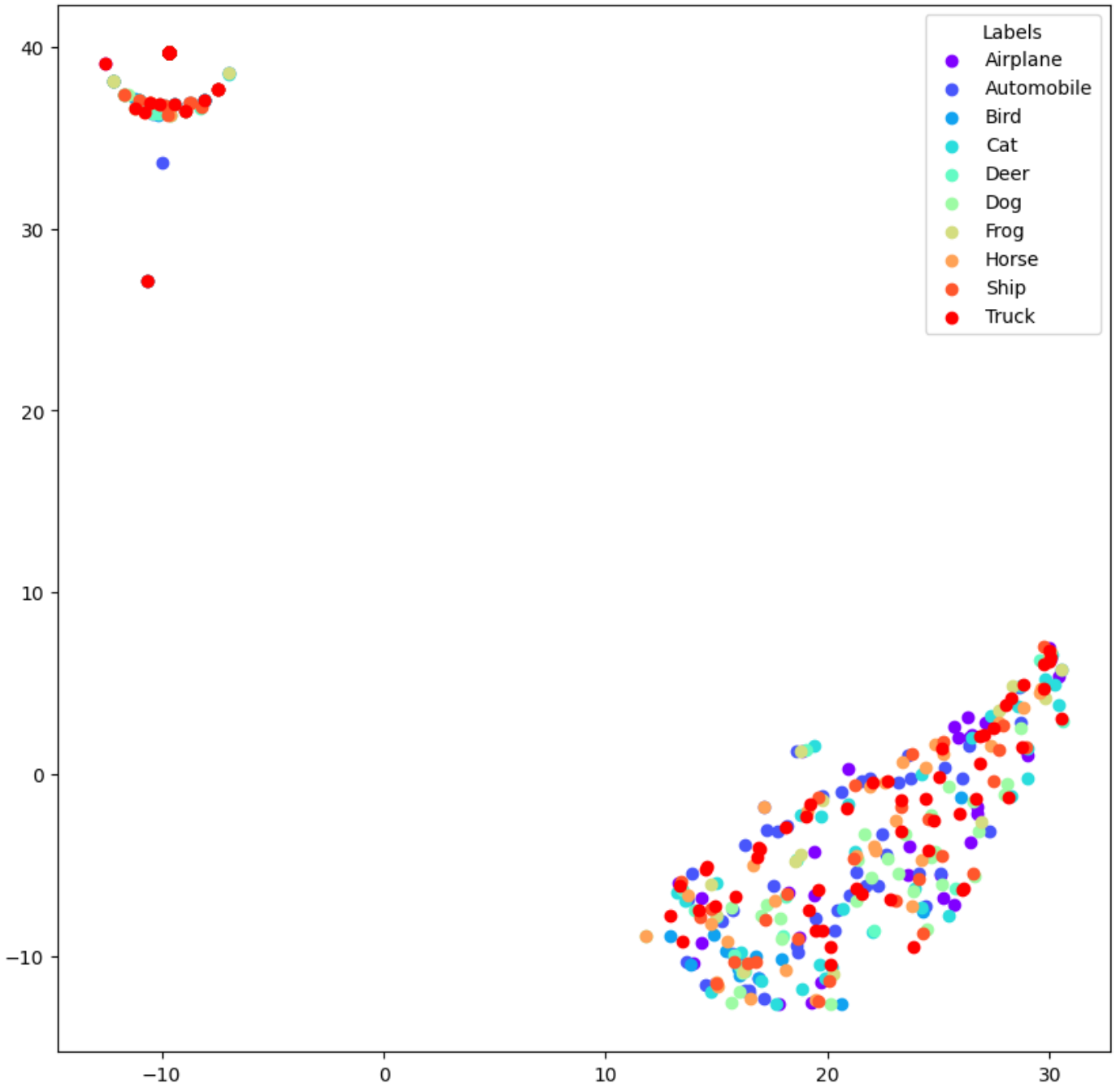

Fig. 36. t-SNE plot for *fc2* layer in VGG16 model.

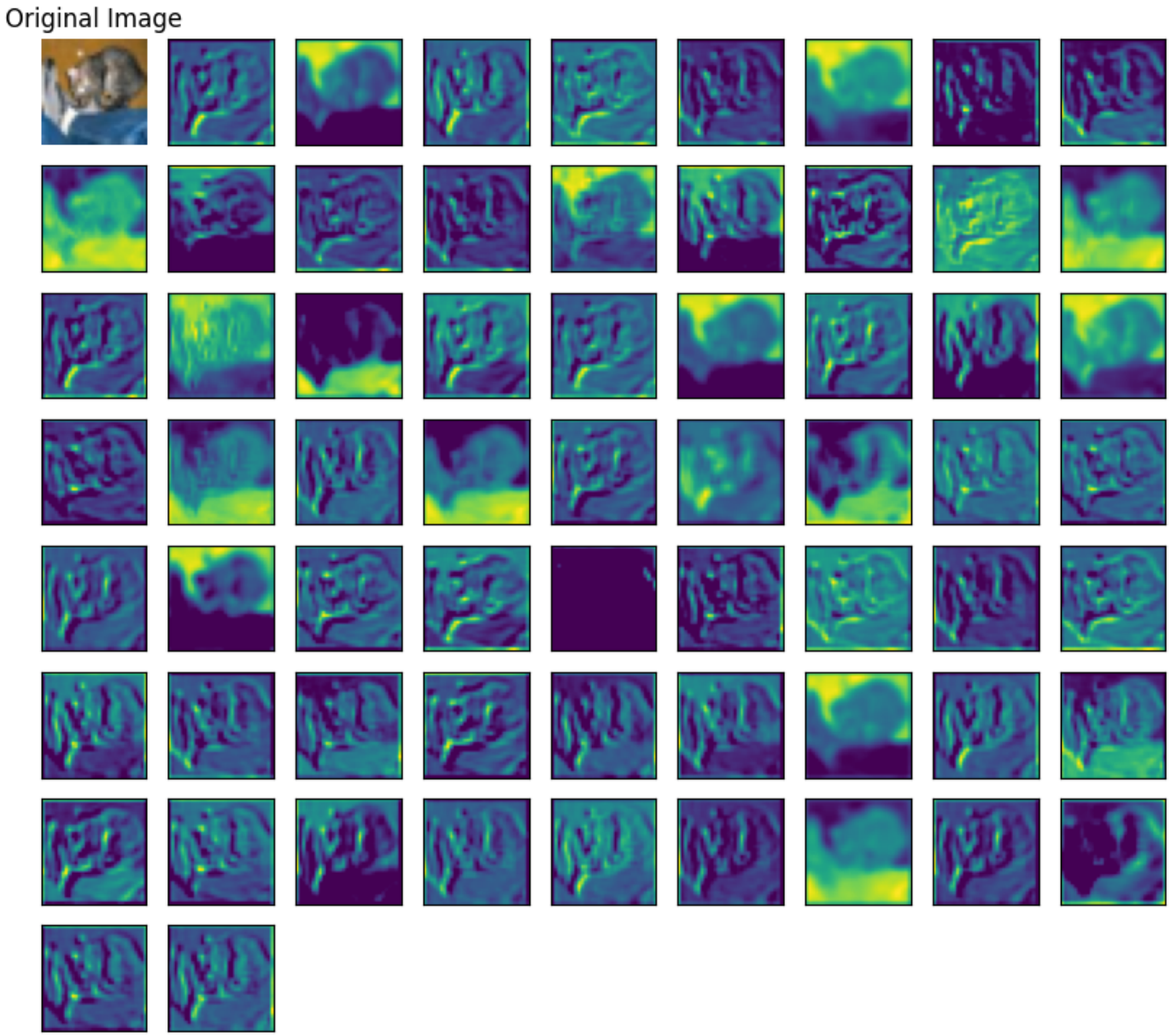

Fig. 29. VGG16 Sample Image 1 Activations - Cat.