Neural Network Models for Object Recognition

Postgraduate Diploma in Artificial Intelligence
Machine Learning module

Guilherme Amorim

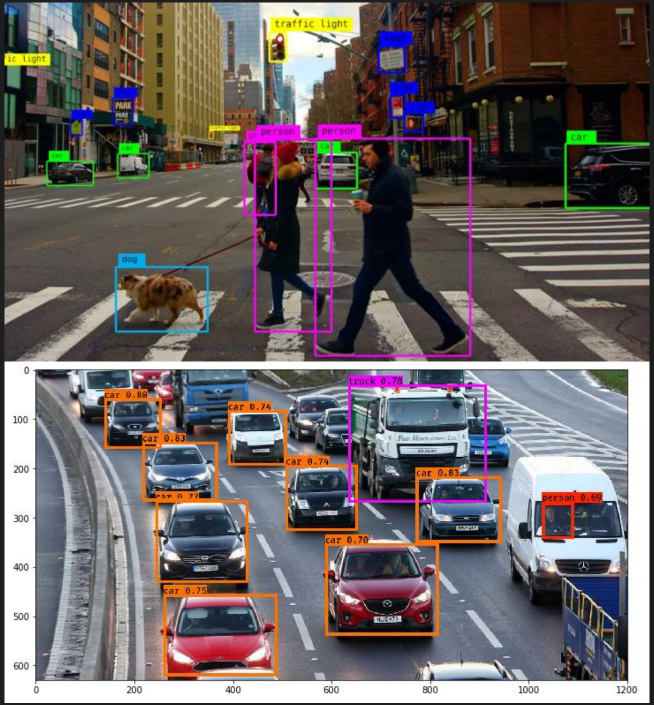
Jan 2025

University of Essex

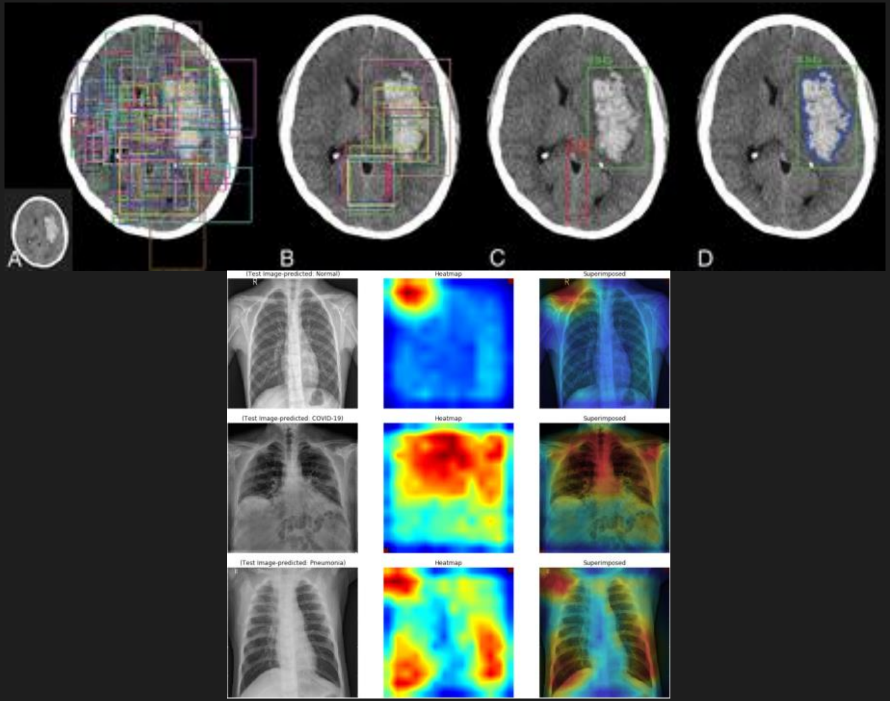
Layout

- Introduction
- Methodology
  - Dataset description
  - Dataset partitioning
  - Network architecture
  - Loss function
  - Activation functions
  - Optimizer algorithm
  - Training approach
- Results
  - Model performance
  - Troubleshooting
- Discussion
  - Critical analysis
  - Learning outcomes and reflections
- Conclusion
- References

Introduction – neural networks for object recognition



Autonomous vehicles

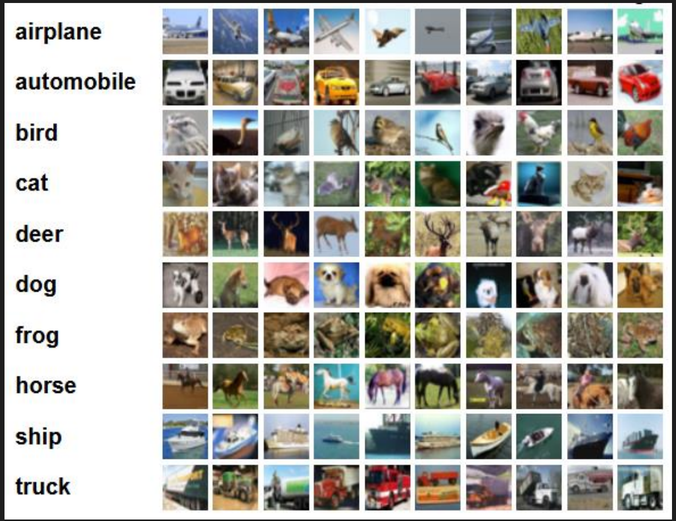

A
B
C
D
(Test image-predicted: Normal)
Heatmap
Superimposed
(Test image-predicted: COVID-19)
Heatmap
Superimposed
(Test image-predicted: Pneumonia)
Heatmap
Superimposed

Medical imaging

References: Chang et al., 2018; Sarki et al., 2022; Augmented AI, 2023; ubiAI, 2023

Dataset

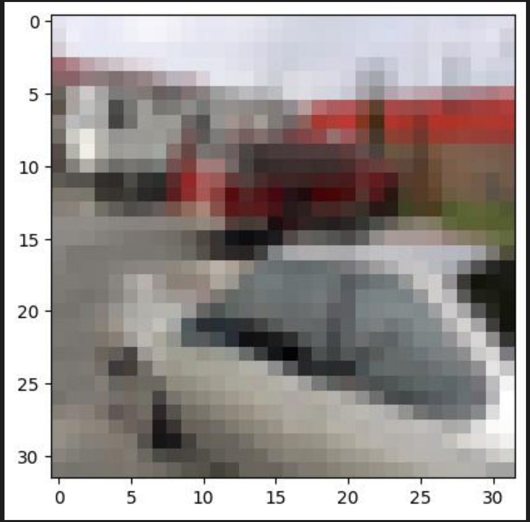
CIFAR-10 dataset (Canadian Institute For Advanced Research Dataset)



CIFAR-10 categories

32 pixels

32 pixels

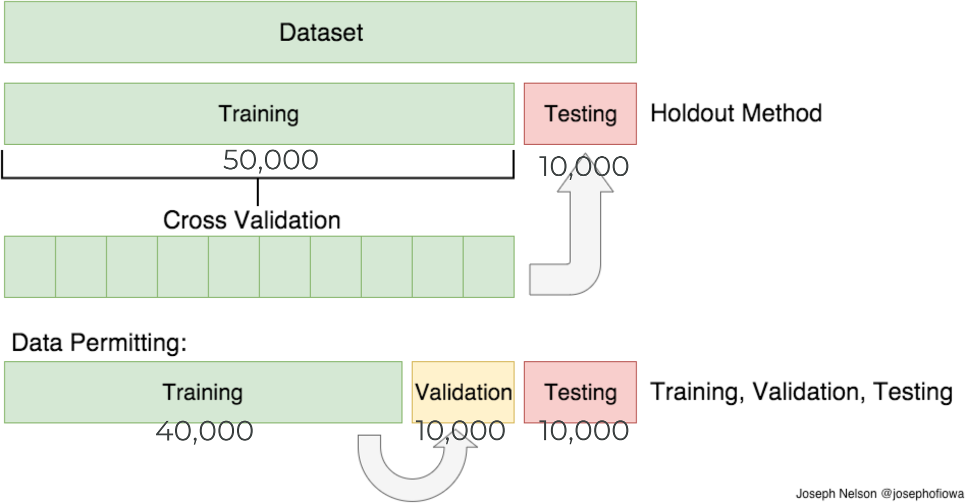


Example image

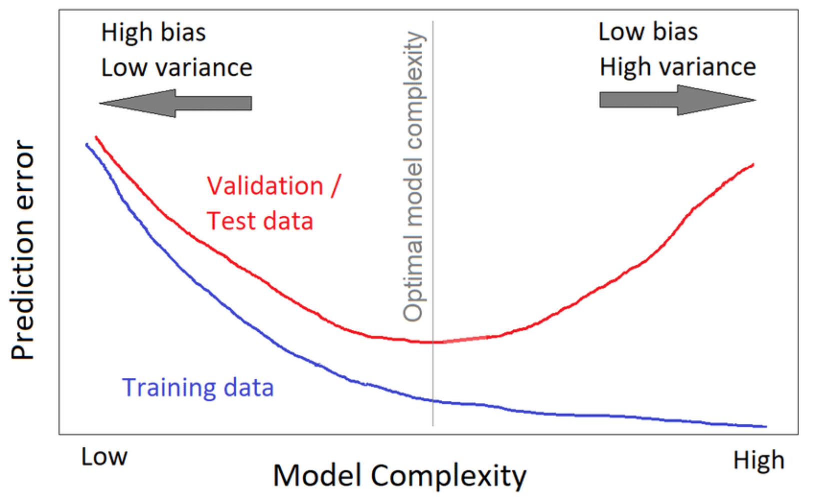
Pre-processing: data normalisation (scaling pixel values: 0-255 to 0-1)

References: Krizhevsky, 2009a, 2009b

Dataset partitioning



Train-test-validate split



Bias-variance tradeoff

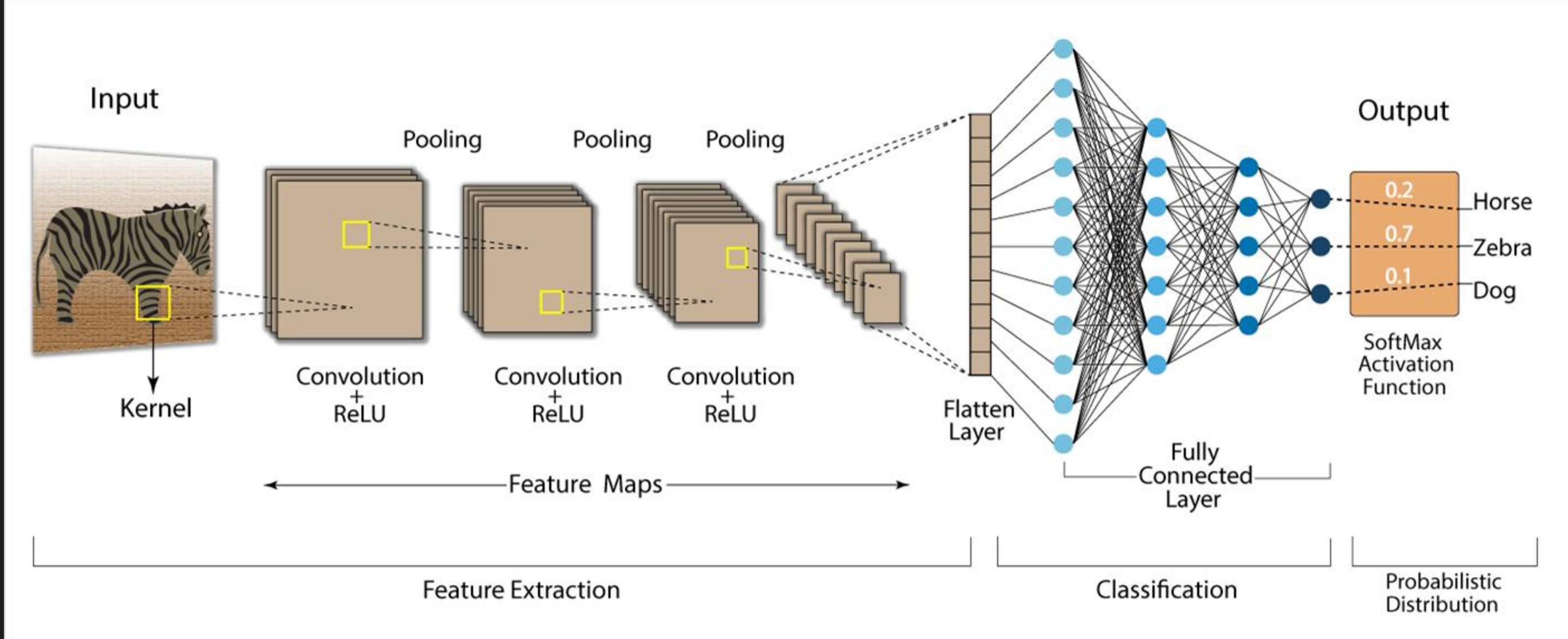
```
# transforming dataset into dataframe

(x_train_all, y_train_all), (x_test, y_test) = cifar10.load_data()
```

```
# creating the validation set
VALIDATION_SIZE = 10000

x_val = x_train_all[:VALIDATION_SIZE]
y_val = y_train_all[:VALIDATION_SIZE]
print("Image validation set shape:\n", x_val.shape)
print("Labels validation set shape:\n", y_val.shape)
```

```
# removing validation set instances from training set
x_train = x_train_all[VALIDATION_SIZE:]
y_train= y_train_all[VALIDATION_SIZE:]
print("Image training set shape:\n", x_train.shape)
print("Labels training set shape:\n", y_train.shape)
```

References: Berrar, 2019; Bronshtein, 2020; Keras: Deep Learning for humans, no date; Natras, Soja and Schmidt, 2022

Network architecture



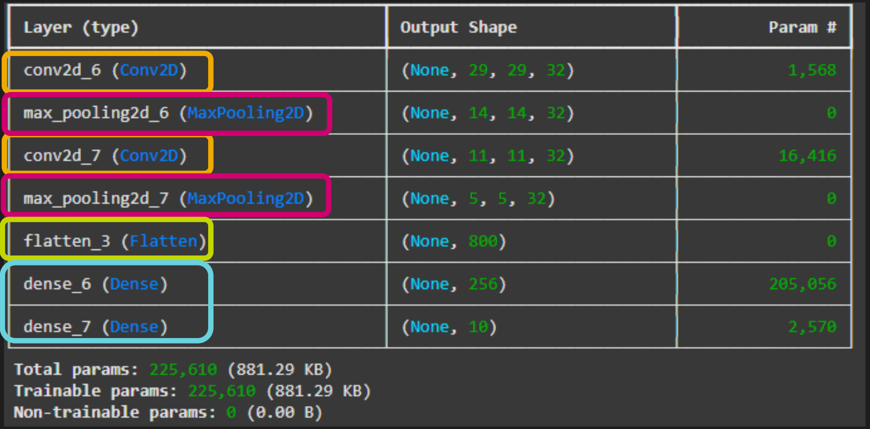
Convolutional layers

Dense (fully-connected) layers

References: Krizhevsky, Suctskever and Hinton, 2012; Swapna., 2020

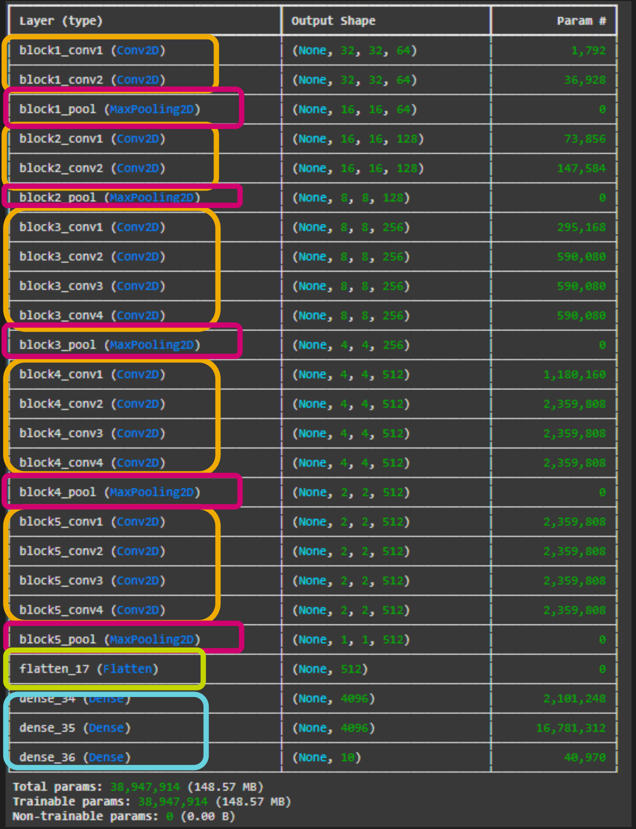
Network architecture

Simple CNN



```
Layer (type)                  Output Shape             Param #
conv2d_6 (Conv2D)             (None, 29, 29, 32)       1,568
max_pooling2d_6 (MaxPooling2D) (None, 14, 14, 32)      0
conv2d_7 (Conv2D)             (None, 11, 11, 32)       16,416
max_pooling2d_7 (MaxPooling2D) (None, 5, 5, 32)        0
flatten_3 (Flatten)           (None, 800)              0
dense_6 (Dense)               (None, 256)              205,056
dense_7 (Dense)               (None, 10)               2,570
Total params: 225,610 (881.29 KB)
Trainable params: 225,610 (881.29 KB)
Non-trainable params: 0 (0.00 B)
```

Visual Geometry Group (VGG)-19



```
Layer (type)                  Output Shape             Param #
block1_conv1 (Conv2D)         (None, 32, 32, 64)       1,792
block1_conv2 (Conv2D)         (None, 32, 32, 64)       36,928
block1_pool (MaxPooling2D)    (None, 16, 16, 64)       0
block2_conv1 (Conv2D)         (None, 16, 16, 128)      73,856
block2_conv2 (Conv2D)         (None, 16, 16, 128)      147,584
block2_pool (MaxPooling2D)    (None, 8, 8, 128)        0
block3_conv1 (Conv2D)         (None, 8, 8, 256)        295,168
block3_conv2 (Conv2D)         (None, 8, 8, 256)        590,080
block3_conv3 (Conv2D)         (None, 8, 8, 256)        590,080
block3_conv4 (Conv2D)         (None, 8, 8, 256)        590,080
block3_pool (MaxPooling2D)    (None, 4, 4, 256)        0
block4_conv1 (Conv2D)         (None, 4, 4, 512)        1,180,160
block4_conv2 (Conv2D)         (None, 4, 4, 512)        2,359,808
block4_conv3 (Conv2D)         (None, 4, 4, 512)        2,359,808
block4_conv4 (Conv2D)         (None, 4, 4, 512)        2,359,808
block4_pool (MaxPooling2D)    (None, 2, 2, 512)        0
block5_conv1 (Conv2D)         (None, 2, 2, 512)        2,359,808
block5_conv2 (Conv2D)         (None, 2, 2, 512)        2,359,808
block5_conv3 (Conv2D)         (None, 2, 2, 512)        2,359,808
block5_conv4 (Conv2D)         (None, 2, 2, 512)        2,359,808
block5_pool (MaxPooling2D)    (None, 1, 1, 512)        0
flatten_17 (Flatten)          (None, 512)              0
dense_34 (Dense)              (None, 4096)             2,101,248
dense_35 (Dense)              (None, 4096)             16,781,312
dense_36 (Dense)              (None, 10)               40,970
Total params: 38,947,914 (148.57 MB)
Trainable params: 38,947,914 (148.57 MB)
Non-trainable params: 0 (0.00 B)
```

Simple CNN:

2 convolutional layers
2 pooling layers (max)
1 flattening layer
2 dense layers
225,610 parameters

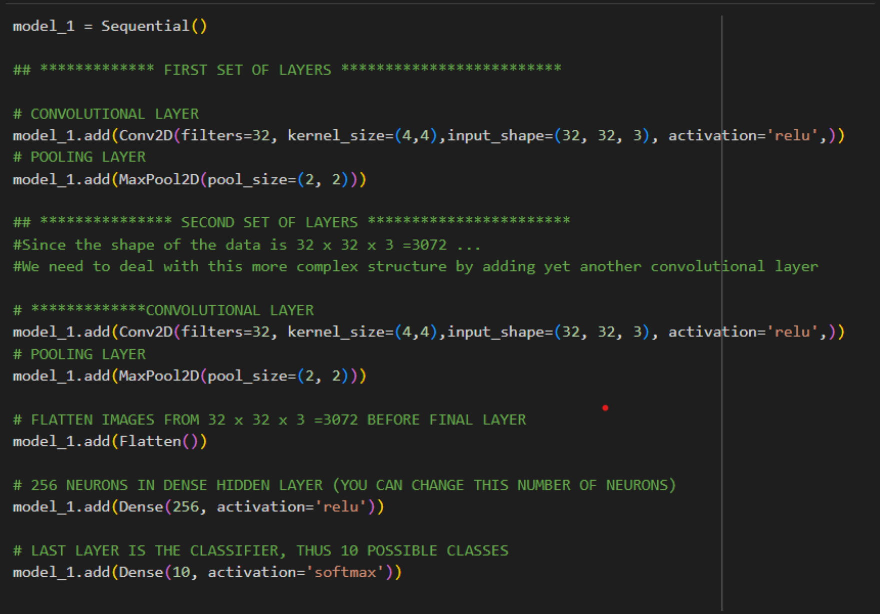
VGG-19:

16 convolutional layers
5 pooling layers (max)
1 flattening layer
3 dense layers
38,947,914 parameters

References: Simonyan and Zisserman, 2015

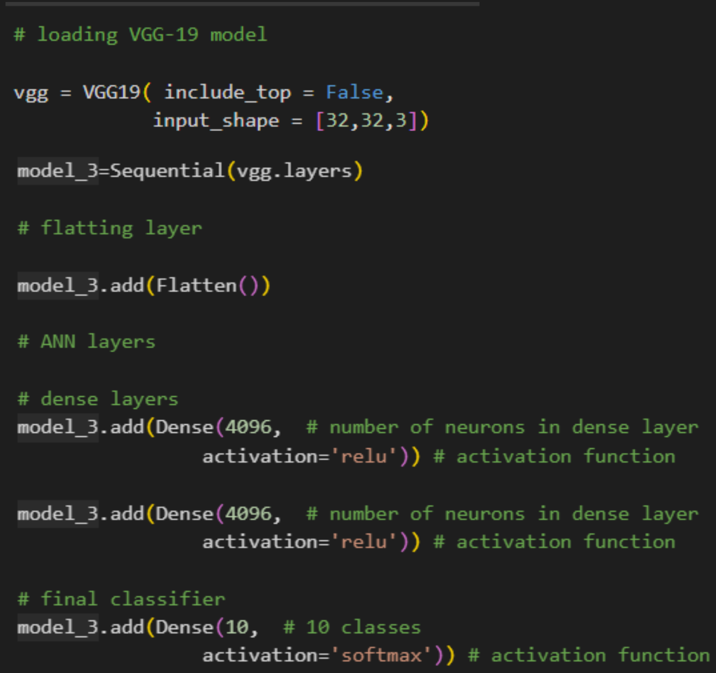
Network architecture

Simple CNN:



```
model_1 = Sequential()

## ************* FIRST SET OF LAYERS *************************

# CONVOLUTIONAL LAYER
model_1.add(Conv2D(filters=32, kernel_size=(4,4),input_shape=(32, 32, 3), activation='relu',))
# POOLING LAYER
model_1.add(MaxPool2D(pool_size=(2, 2)))

## *************** SECOND SET OF LAYERS ***********************
#Since the shape of the data is 32 x 32 x 3 =3072 ...
#We need to deal with this more complex structure by adding yet another convolutional layer

# *************CONVOLUTIONAL LAYER
model_1.add(Conv2D(filters=32, kernel_size=(4,4),input_shape=(32, 32, 3), activation='relu',))
# POOLING LAYER
model_1.add(MaxPool2D(pool_size=(2, 2)))

# FLATTEN IMAGES FROM 32 x 32 x 3 =3072 BEFORE FINAL LAYER
model_1.add(Flatten())

# 256 NEURONS IN DENSE HIDDEN LAYER (YOU CAN CHANGE THIS NUMBER OF NEURONS)
model_1.add(Dense(256, activation='relu'))

# LAST LAYER IS THE CLASSIFIER, THUS 10 POSSIBLE CLASSES
model_1.add(Dense(10, activation='softmax'))
```

VGG-19:

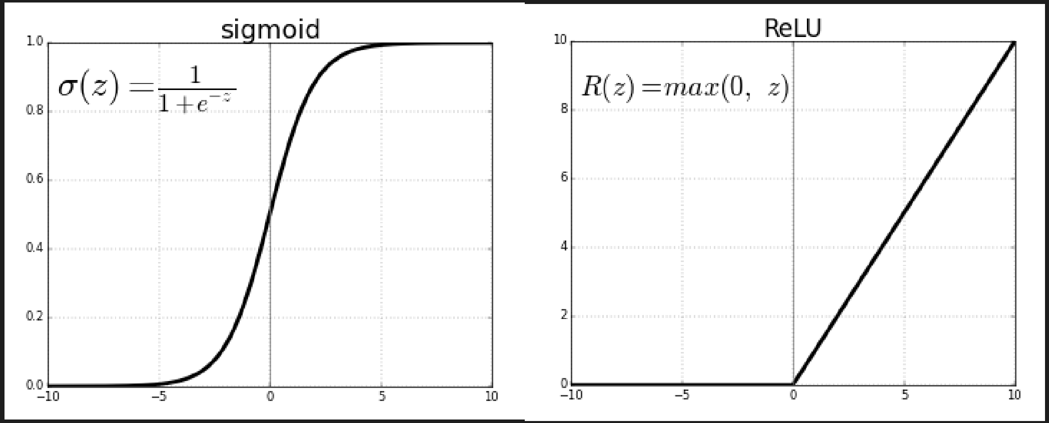


```
# loading VGG-19 model

vgg = VGG19( include_top = False,
            input_shape = [32,32,3])

model_3=Sequential(vgg.layers)

# flatting layer

model_3.add(Flatten())

# ANN layers

# dense layers
model_3.add(Dense(4096,  # number of neurons in dense layer
                  activation='relu')) # activation function

model_3.add(Dense(4096,  # number of neurons in dense layer
                  activation='relu')) # activation function

# final classifier
model_3.add(Dense(10,  # 10 classes
                  activation='softmax')) # activation function
```

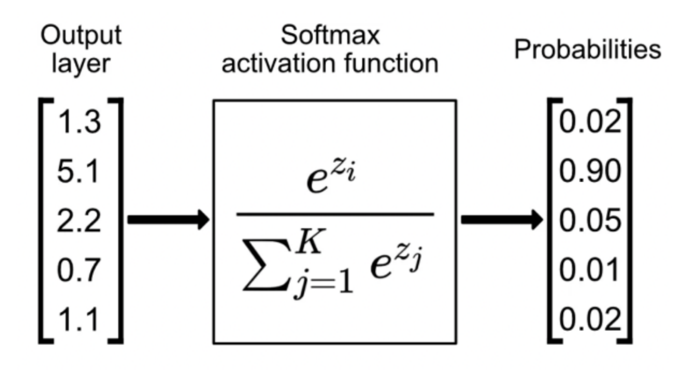
References: Simonyan and Zisserman, 2015

Activation functions

Activation function within network:
Rectifier Linear Unit (ReLU)



Final activation function:
softmax


Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

References: Bag, 2023; Belagatti, 2024; Srivastava, 2024

Loss function

Sparse categorical cross-entropy (log) loss

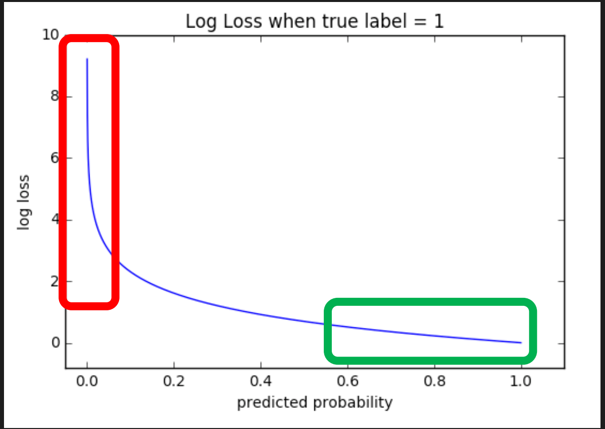
$$L = -\sum_{k=1}^{K} \boxed{y_k} \log \boxed{(p_k)}$$

y_k: true probability of class k
(1 for the correct class,
0 for all others)

p_k: predicted probability of class k (0-1)

```
model_1.compile(loss='sparse_categorical_crossentropy',
                optimizer='sgd',
                metrics=['accuracy'])
```

High loss for incorrect predictions (p_k ~0 and y_k = 1)



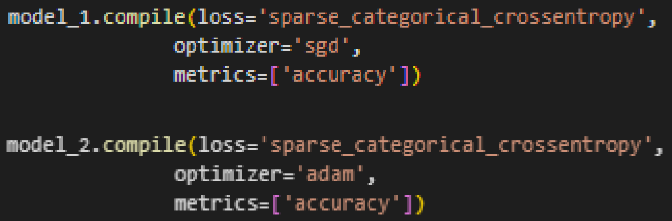
Low loss for correct predictions (p_k~1, y_k=1)

References: Fortuner, 2017; Hughes, 2024

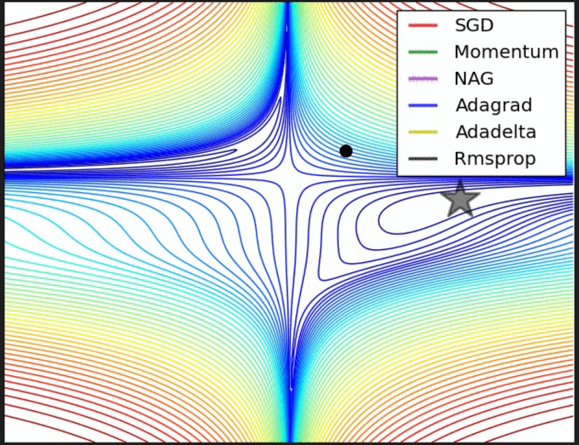
Optimizer algorithm

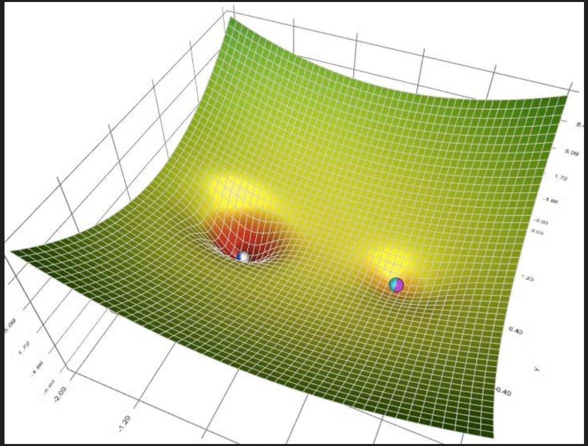
Examples:

- **Stochastic gradient descent (SGD)**
- SGD with momentum
- AdaGrad (Adaptive Gradient)
- RMSProp (Root Mean Square Propagation)
- **Adam (Adaptive Moment Estimation)**



```
model_1.compile(loss='sparse_categorical_crossentropy',
                optimizer='sgd',
                metrics=['accuracy'])

model_2.compile(loss='sparse_categorical_crossentropy',
                optimizer='adam',
                metrics=['accuracy'])
```

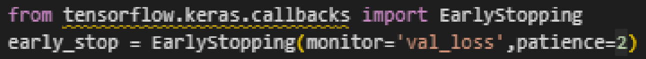




References: Adhikari, 2023; Giordano 2020; Jiang, 2020; Kingma and Ba, 2017; Lu, 2017; Musstafa, 2022; Wilson et al., 2018

Training approach

- Batch size: 64
- Maximum training epochs: 50
- Callback/early stop: if validation loss increases for 2 consecutive epochs
- Learning rate (optimizer algorithm defaults):
  - SGD: 0.01 (1e-2)
  - Adam: 0.001 (1e-3)



```
from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor='val_loss',patience=2)
```

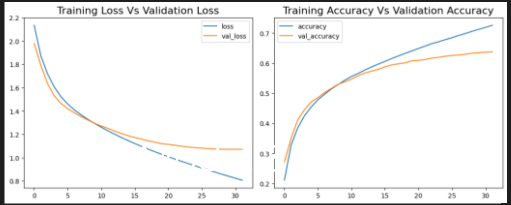
References: Chang and Pathak, 2020

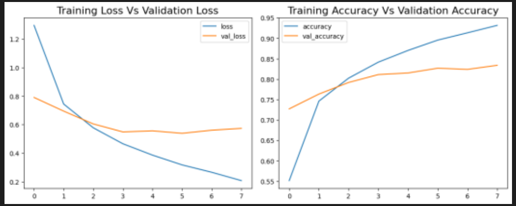
Model performance

*Until early stopping rule is met
Models ran on Google Colab Pro default environment and A100 GPU hardware accelerator

Model	Architecture	Optimizer	Epochs*	Time	Training accuracy	Training loss	Validation accuracy	Validation loss
1	Simple CNN	SGD	33	55 s	0.72	0.80	0.64	1.06
2	Simple CNN	Adam	9	18 s	0.78	0.65	0.64	1.15

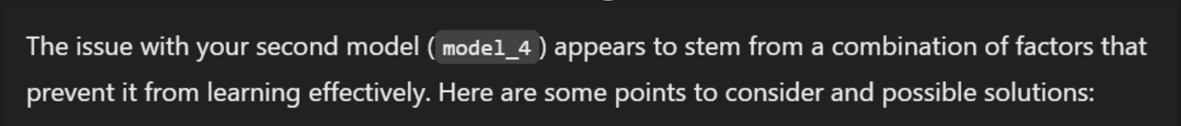
Model 1



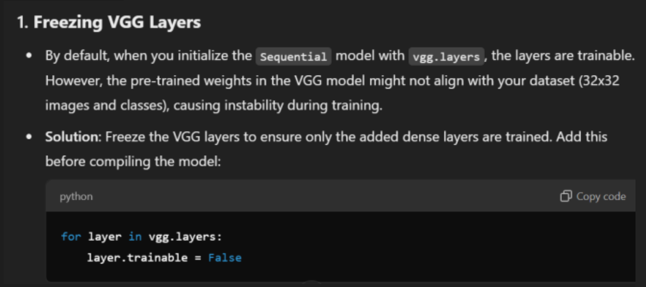


Model 2

Troubleshooting - ChatGPT



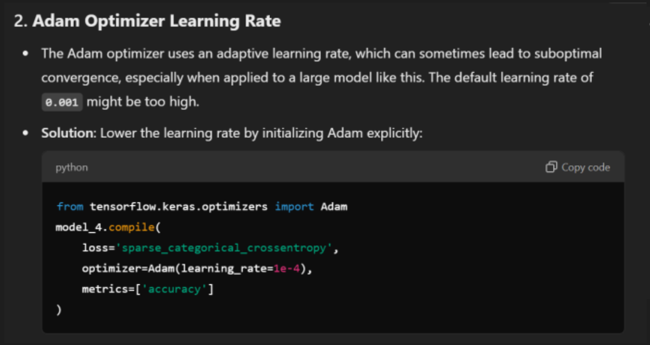
The issue with your second model (`model_4`) appears to stem from a combination of factors that prevent it from learning effectively. Here are some points to consider and possible solutions:



1. **Freezing VGG Layers**

- By default, when you initialize the `Sequential` model with `vgg.layers`, the layers are trainable. However, the pre-trained weights in the VGG model might not align with your dataset (32x32 images and classes), causing instability during training.
- **Solution:** Freeze the VGG layers to ensure only the added dense layers are trained. Add this before compiling the model:

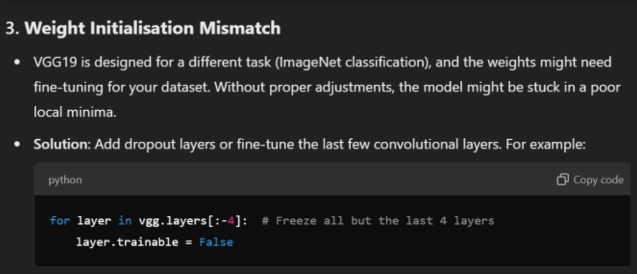
```python
for layer in vgg.layers:
    layer.trainable = False
```



2. **Adam Optimizer Learning Rate**

- The Adam optimizer uses an adaptive learning rate, which can sometimes lead to suboptimal convergence, especially when applied to a large model like this. The default learning rate of `0.001` might be too high.
- **Solution:** Lower the learning rate by initializing Adam explicitly:

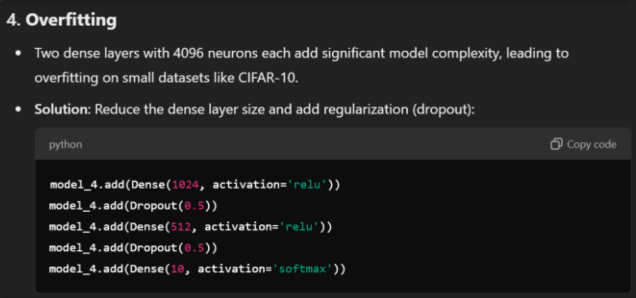
```python
from tensorflow.keras.optimizers import Adam
model_4.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=Adam(learning_rate=1e-4),
    metrics=['accuracy']
)
```



3. **Weight Initialisation Mismatch**

- VGG19 is designed for a different task (ImageNet classification), and the weights might need fine-tuning for your dataset. Without proper adjustments, the model might be stuck in a poor local minima.
- **Solution:** Add dropout layers or fine-tune the last few convolutional layers. For example:

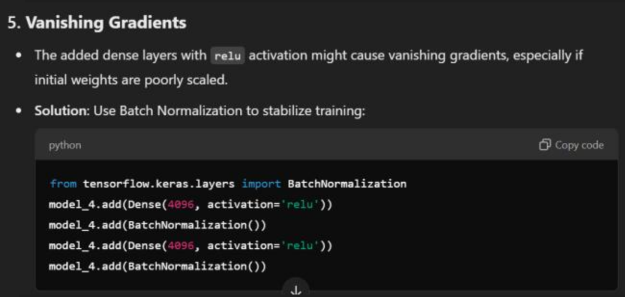
```python
for layer in vgg.layers[:-4]:  # Freeze all but the last 4 layers
    layer.trainable = False
```



4. **Overfitting**

- Two dense layers with 4096 neurons each add significant model complexity, leading to overfitting on small datasets like CIFAR-10.
- **Solution:** Reduce the dense layer size and add regularization (dropout):

```python
model_4.add(Dense(1024, activation='relu'))
model_4.add(Dropout(0.5))
model_4.add(Dense(512, activation='relu'))
model_4.add(Dropout(0.5))
model_4.add(Dense(10, activation='softmax'))
```



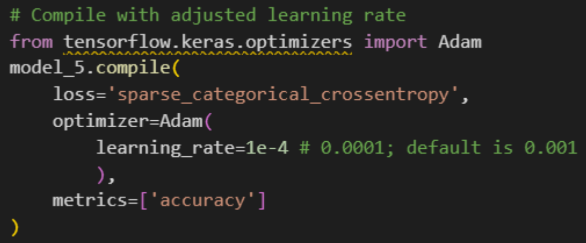
5. **Vanishing Gradients**

- The added dense layers with `relu` activation might cause vanishing gradients, especially if initial weights are poorly scaled.
- **Solution:** Use Batch Normalization to stabilize training:

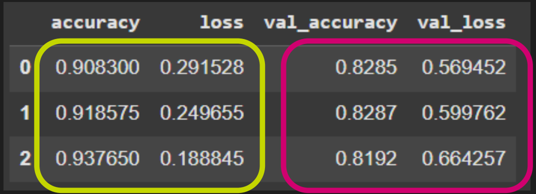
```python
from tensorflow.keras.layers import BatchNormalization
model_4.add(Dense(4096, activation='relu'))
model_4.add(BatchNormalization())
model_4.add(Dense(4096, activation='relu'))
model_4.add(BatchNormalization())
```

Model tuning – learning rate (model 5)

Aim – stabilise learning with smaller learning rate = 1e-4

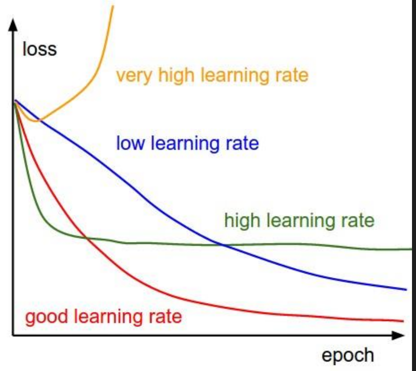


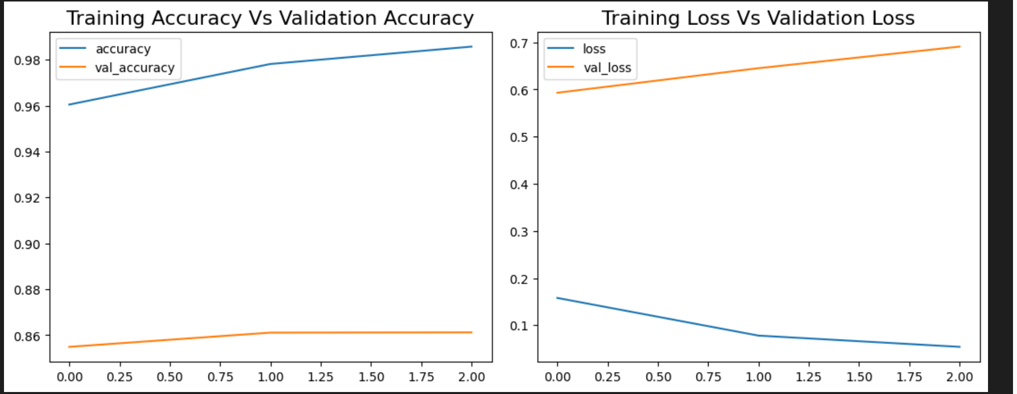
```
# Compile with adjusted learning rate
from tensorflow.keras.optimizers import Adam
model_5.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=Adam(
        learning_rate=1e-4 # 0.0001; default is 0.001
        ),
    metrics=['accuracy']
)
```

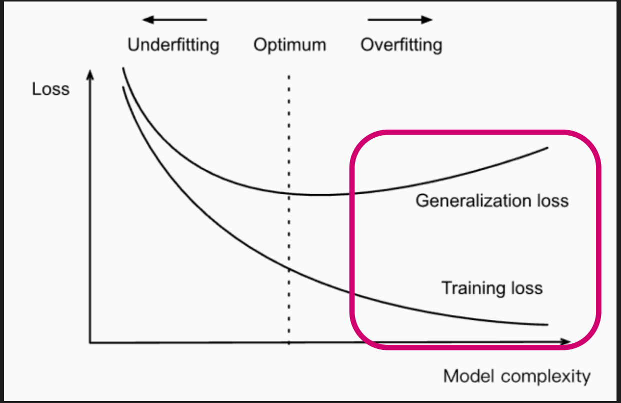


	accuracy	loss	val_accuracy	val_loss
0	0.908300	0.291528	0.8285	0.569452
1	0.918575	0.249655	0.8287	0.599762
2	0.937650	0.188845	0.8192	0.664257

Training time: 28 s





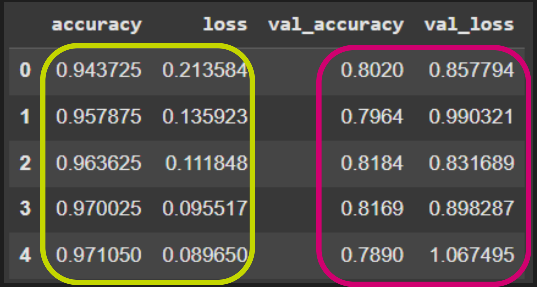


References: Zulkifli, 2018; Kurban, 2021

Model tuning – batch normalisation (model 6)

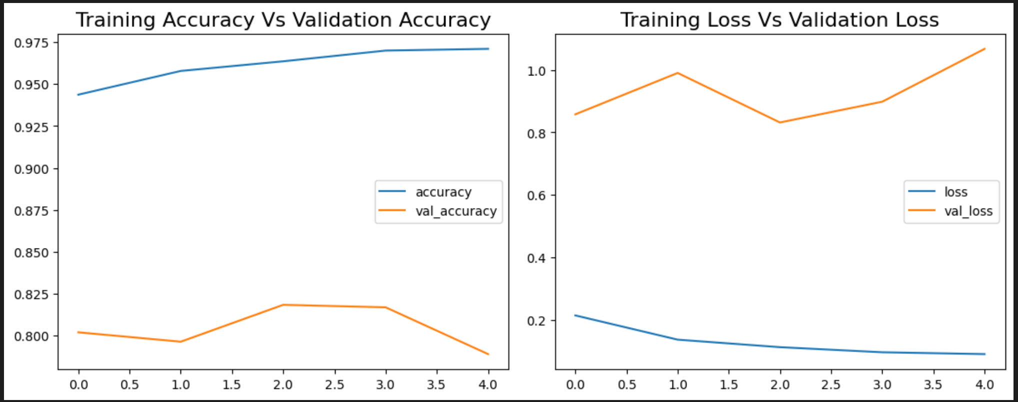
Aim – normalise outputs across different filters within the same batch (avoid vanishing gradients)

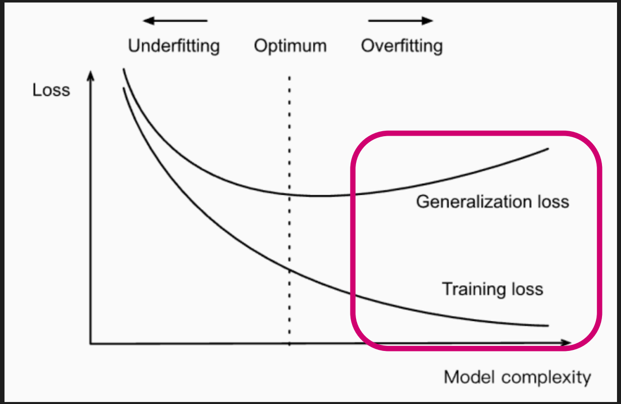
```
# Flatten and add dense layers + batch normalisation

model_6.add(Flatten())
model_6.add(Dense(4096,  activation='relu'))
model_6.add(BatchNormalization())
model_6.add(Dense(4096,  activation='relu'))
model_6.add(BatchNormalization())
model_6.add(Dense(10,  activation='softmax'))
```



	accuracy	loss	val_accuracy	val_loss
0	0.943725	0.213584	0.8020	0.857794
1	0.957875	0.135923	0.7964	0.990321
2	0.963625	0.111848	0.8184	0.831689
3	0.970025	0.095517	0.8169	0.898287
4	0.971050	0.089650	0.7890	1.067495

Training time: 44 s



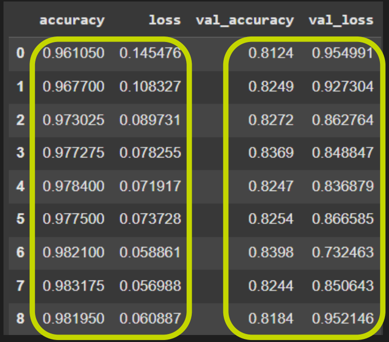


References: Kurban, 2021

Model tuning – regularisation I (model 7)

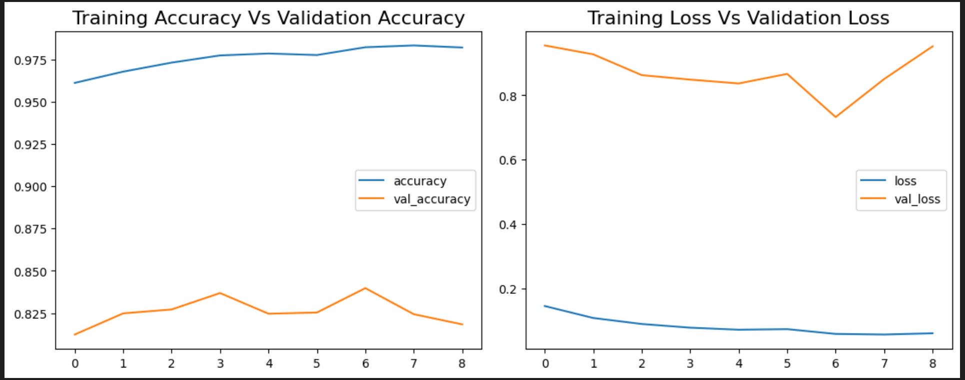
Aim – discard some neurons in dense layers (dropout) to avoid overfitting

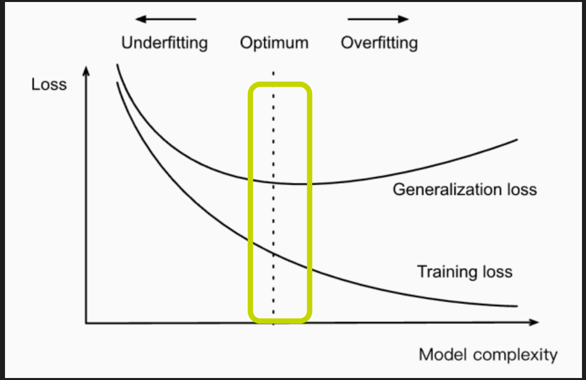
```
# Flatten and add dense layers + batch normalisation + dropout

model_7.add(Flatten())
model_7.add(Dense(4096,  activation='relu'))
model_7.add(BatchNormalization())
model_7.add(Dropout(0.5)) # discard 50% of neurons
model_7.add(Dense(4096,  activation='relu'))
model_7.add(BatchNormalization())
model_7.add(Dropout(0.5)) # discard 50% of neurons
model_7.add(Dense(10,  activation='softmax'))
```



	accuracy	loss	val_accuracy	val_loss
0	0.961050	0.145476	0.8124	0.954991
1	0.967700	0.108327	0.8249	0.927304
2	0.973025	0.089731	0.8272	0.862764
3	0.977275	0.078255	0.8369	0.848847
4	0.978400	0.071917	0.8247	0.836879
5	0.977500	0.073728	0.8254	0.866585
6	0.982100	0.058861	0.8398	0.732463
7	0.983175	0.056988	0.8244	0.850643
8	0.981950	0.060887	0.8184	0.952146

Training time: 1min 11 s

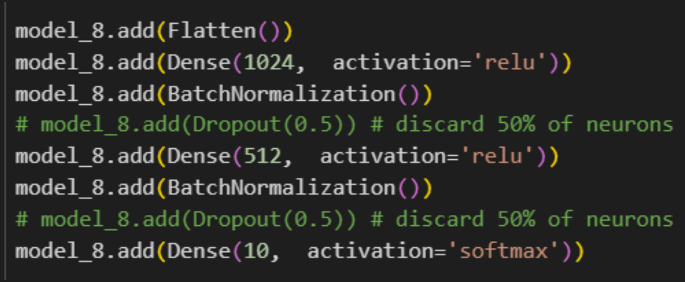




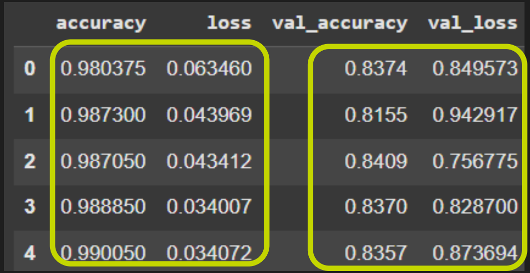
References: Kurban, 2021

Model tuning – regularisation II (model 8)

Aim – make dense layers simpler to avoid overfitting (without dropout)

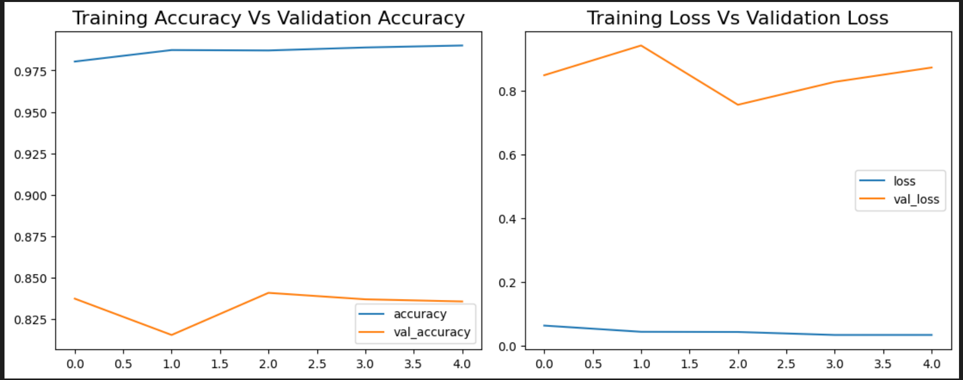


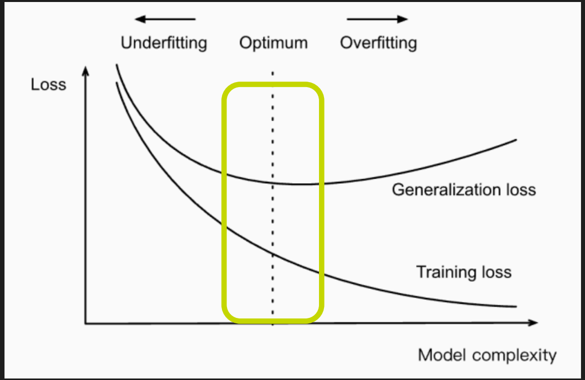
```
model_8.add(Flatten())
model_8.add(Dense(1024,  activation='relu'))
model_8.add(BatchNormalization())
# model_8.add(Dropout(0.5)) # discard 50% of neurons
model_8.add(Dense(512,  activation='relu'))
model_8.add(BatchNormalization())
# model_8.add(Dropout(0.5)) # discard 50% of neurons
model_8.add(Dense(10,  activation='softmax'))
```



	accuracy	loss	val_accuracy	val_loss
0	0.980375	0.063460	0.8374	0.849573
1	0.987300	0.043969	0.8155	0.942917
2	0.987050	0.043412	0.8409	0.756775
3	0.988850	0.034007	0.8370	0.828700
4	0.990050	0.034072	0.8357	0.873694

Training time: 41 s

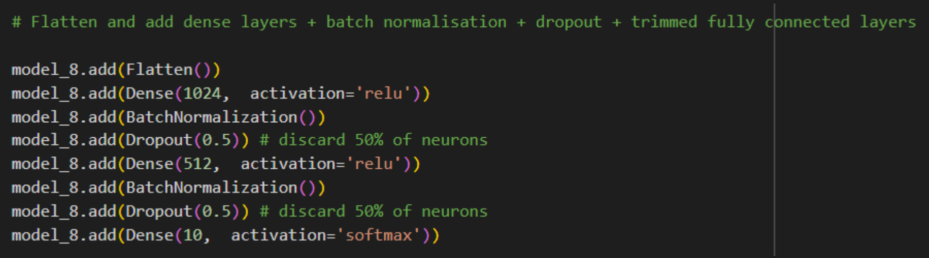




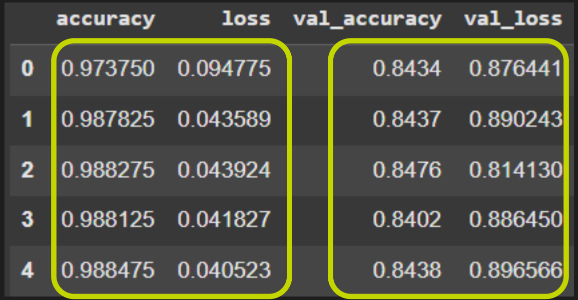
References: Kurban, 2021

Model tuning – regularisation III (model 9)

Aim – make dense layers simpler to avoid overfitting (plus dropout)

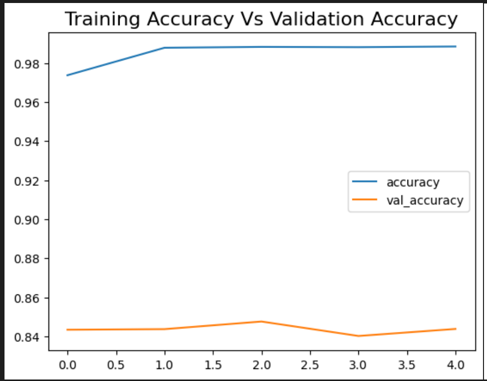


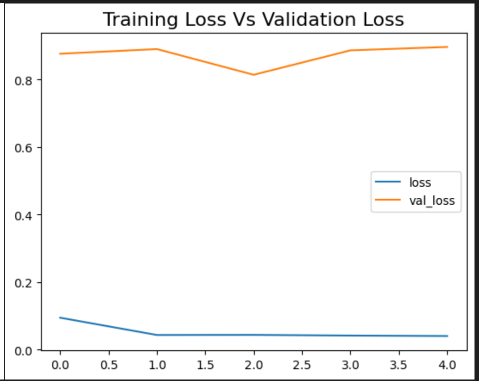
```
# Flatten and add dense layers + batch normalisation + dropout + trimmed fully connected layers

model_8.add(Flatten())
model_8.add(Dense(1024,  activation='relu'))
model_8.add(BatchNormalization())
model_8.add(Dropout(0.5)) # discard 50% of neurons
model_8.add(Dense(512,  activation='relu'))
model_8.add(BatchNormalization())
model_8.add(Dropout(0.5)) # discard 50% of neurons
model_8.add(Dense(10,  activation='softmax'))
```

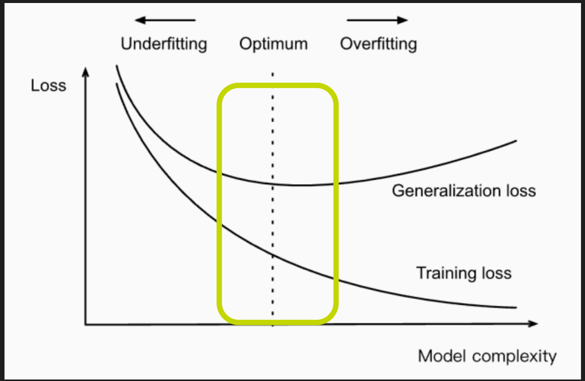


	accuracy	loss	val_accuracy	val_loss
0	0.973750	0.094775	0.8434	0.876441
1	0.987825	0.043589	0.8437	0.890243
2	0.988275	0.043924	0.8476	0.814130
3	0.988125	0.041827	0.8402	0.886450
4	0.988475	0.040523	0.8438	0.896566

Training time: 41 s



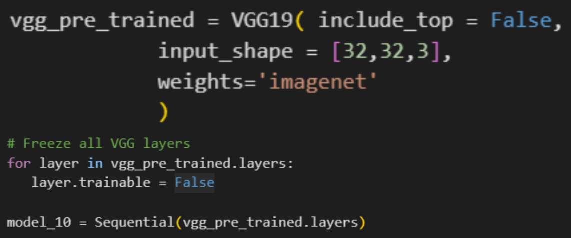




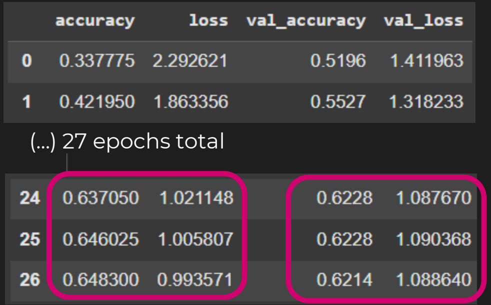
References: Kurban, 2021

Model tuning – pre-trained weights (model 10)

Aim – transfer learning by leveraging pre-trained weights



```
vgg_pre_trained = VGG19( include_top = False,
            input_shape = [32,32,3],
            weights='imagenet'
            )

# Freeze all VGG layers
for layer in vgg_pre_trained.layers:
    layer.trainable = False

model_10 = Sequential(vgg_pre_trained.layers)
```

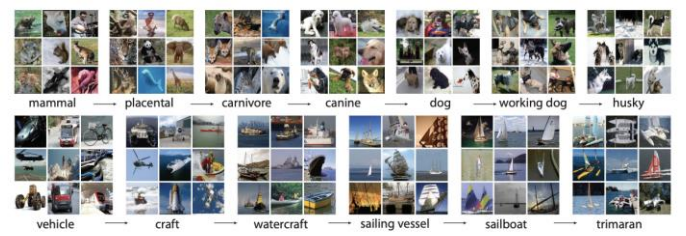


	accuracy	loss	val_accuracy	val_loss
0	0.337775	2.292621	0.5196	1.411963
1	0.421950	1.863356	0.5527	1.318233

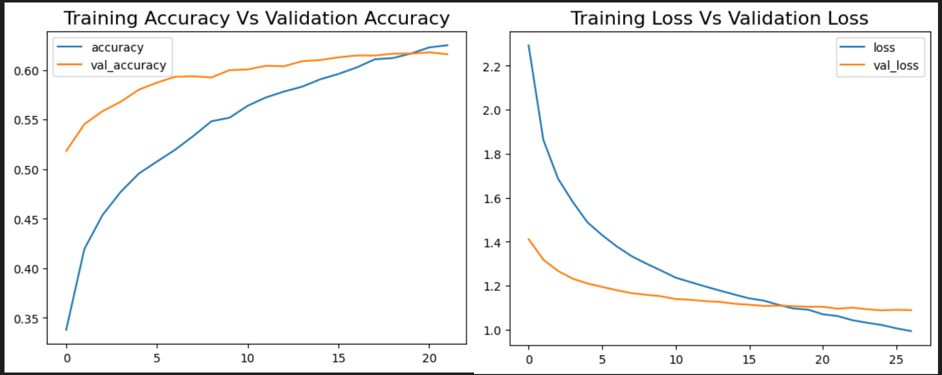
(...) 27 epochs total

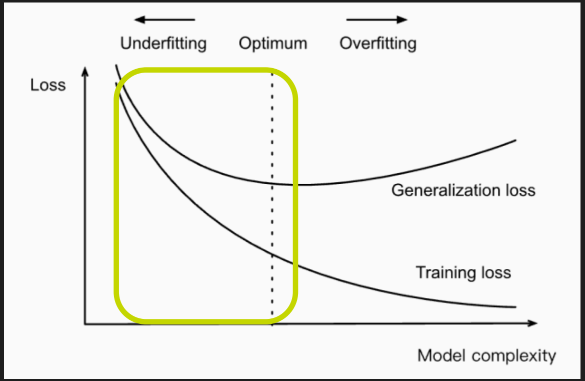
	accuracy	loss	val_accuracy	val_loss
24	0.637050	1.021148	0.6228	1.087670
25	0.646025	1.005807	0.6228	1.090368
26	0.648300	0.993571	0.6214	1.088640

Training time: 1 min 17s


mammal → placental → carnivore → canine → dog → working dog → husky
vehicle → craft → watercraft → sailing vessel → sailboat → trimaran

ImageNet dataset (>14M images, 10,000 classes)

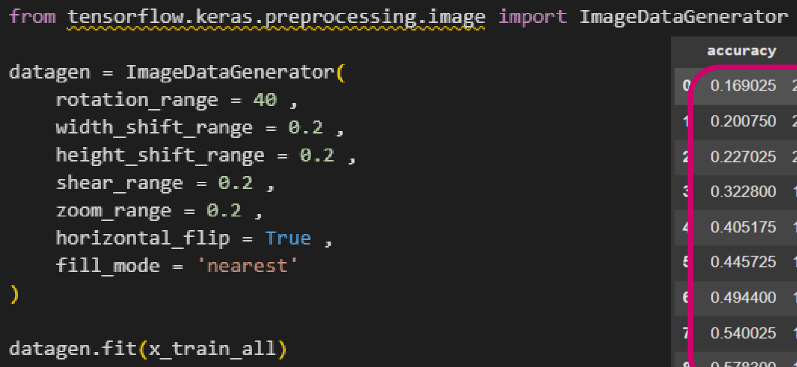




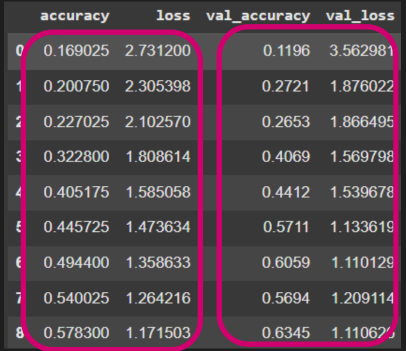
References: Kurban, 2021; Solawetz, 2021; Stanford Vision Lab, Stanford University, Princeton University, 2021

Model tuning - data augmentation (model 11)

Aim – force learning of more representative features by adding noise

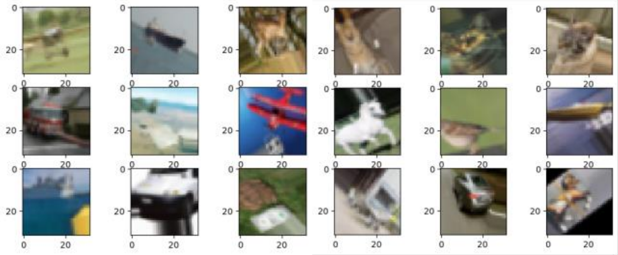


```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range = 40 ,
    width_shift_range = 0.2 ,
    height_shift_range = 0.2 ,
    shear_range = 0.2 ,
    zoom_range = 0.2 ,
    horizontal_flip = True ,
    fill_mode = 'nearest'
)

datagen.fit(x_train_all)
```

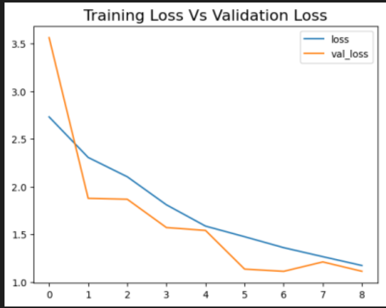


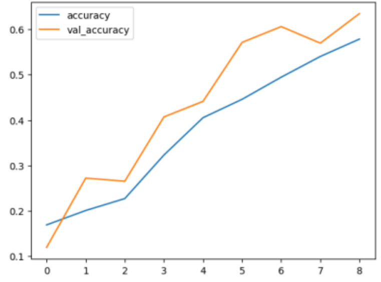
	accuracy	loss	val_accuracy	val_loss
0	0.169025	2.731200	0.1196	3.562981
1	0.200750	2.305398	0.2721	1.876022
2	0.227025	2.102570	0.2653	1.866495
3	0.322800	1.808614	0.4069	1.569798
4	0.405175	1.585058	0.4412	1.539678
5	0.445725	1.473634	0.5711	1.133619
6	0.494400	1.358633	0.6059	1.110129
7	0.540025	1.264216	0.5694	1.209114
8	0.578300	1.171503	0.6345	1.11062

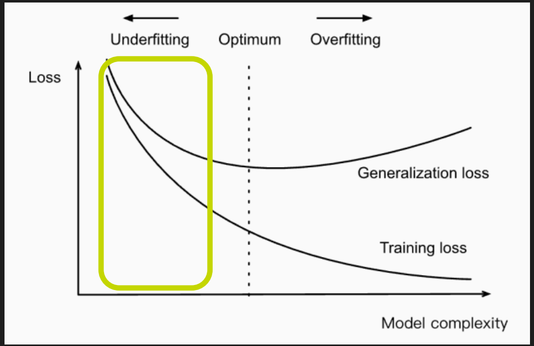
Training time: 4 min 13 s



Example augmented images

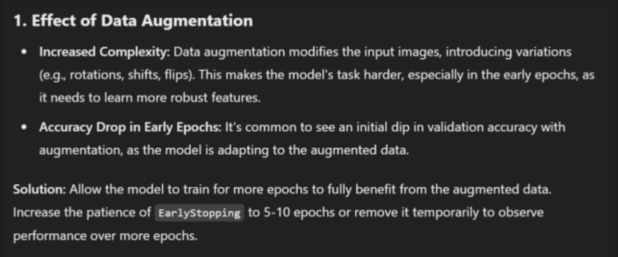






References: Kurban, 2021

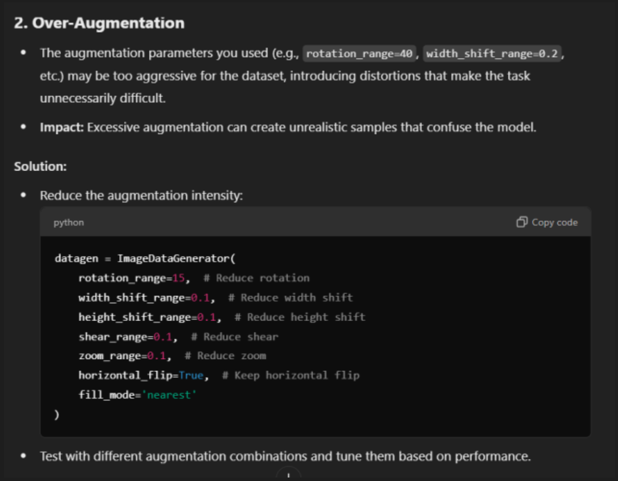
Troubleshooting - ChatGPT



1. Effect of Data Augmentation

- **Increased Complexity:** Data augmentation modifies the input images, introducing variations (e.g., rotations, shifts, flips). This makes the model's task harder, especially in the early epochs, as it needs to learn more robust features.
- **Accuracy Drop in Early Epochs:** It's common to see an initial dip in validation accuracy with augmentation, as the model is adapting to the augmented data.

Solution: Allow the model to train for more epochs to fully benefit from the augmented data. Increase the patience of `EarlyStopping` to 5-10 epochs or remove it temporarily to observe performance over more epochs.



2. Over-Augmentation

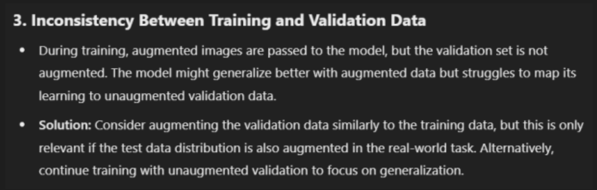
- The augmentation parameters you used (e.g., `rotation_range=40`, `width_shift_range=0.2`, etc.) may be too aggressive for the dataset, introducing distortions that make the task unnecessarily difficult.
- **Impact:** Excessive augmentation can create unrealistic samples that confuse the model.

Solution:

- Reduce the augmentation intensity:

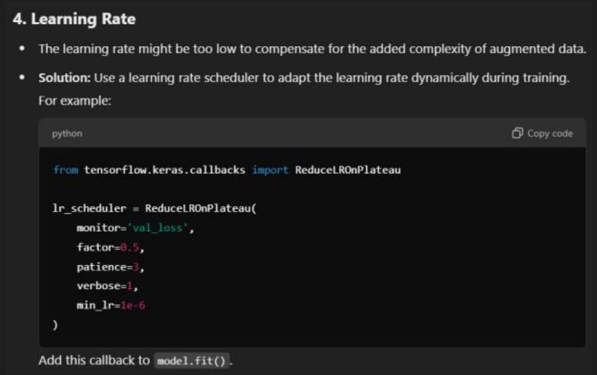
```python
datagen = ImageDataGenerator(
    rotation_range=15,  # Reduce rotation
    width_shift_range=0.1,  # Reduce width shift
    height_shift_range=0.1,  # Reduce height shift
    shear_range=0.1,  # Reduce shear
    zoom_range=0.1,  # Reduce zoom
    horizontal_flip=True,  # Keep horizontal flip
    fill_mode='nearest'
)
```

- Test with different augmentation combinations and tune them based on performance.



3. Inconsistency Between Training and Validation Data

- During training, augmented images are passed to the model, but the validation set is not augmented. The model might generalize better with augmented data but struggles to map its learning to unaugmented validation data.
- **Solution:** Consider augmenting the validation data similarly to the training data, but this is only relevant if the test data distribution is also augmented in the real-world task. Alternatively, continue training with unaugmented validation to focus on generalization.

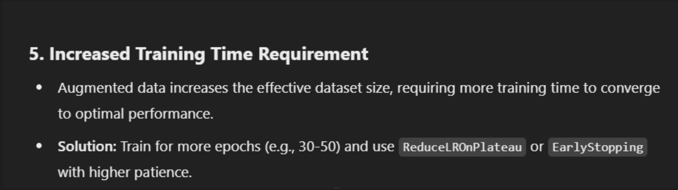


4. Learning Rate

- The learning rate might be too low to compensate for the added complexity of augmented data.
- **Solution:** Use a learning rate scheduler to adapt the learning rate dynamically during training. For example:

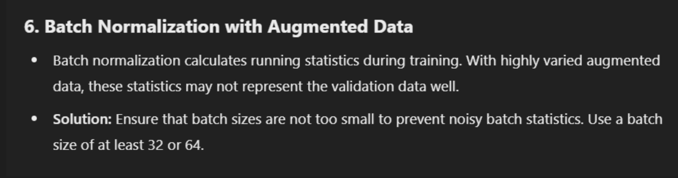
```python
from tensorflow.keras.callbacks import ReduceLROnPlateau

lr_scheduler = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=3,
    verbose=1,
    min_lr=1e-6
)
```

Add this callback to `model.fit()`.



5. Increased Training Time Requirement

- Augmented data increases the effective dataset size, requiring more training time to converge to optimal performance.
- **Solution:** Train for more epochs (e.g., 30-50) and use `ReduceLROnPlateau` or `EarlyStopping` with higher patience.

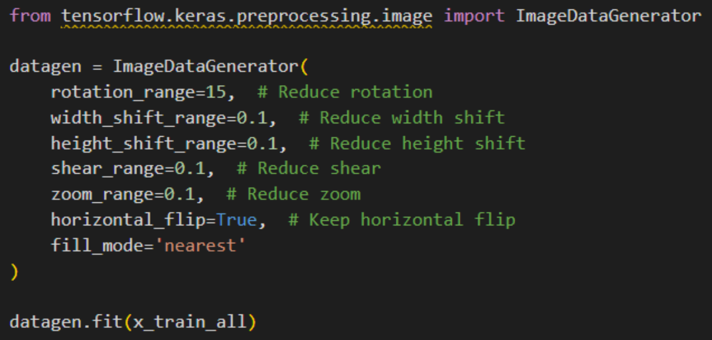


6. Batch Normalization with Augmented Data

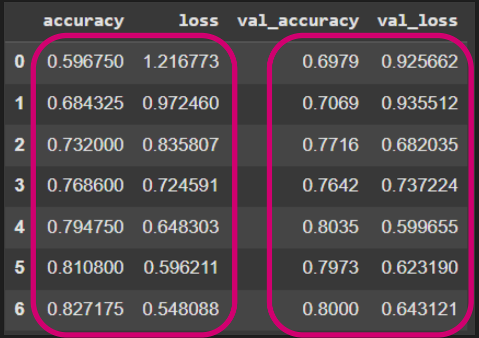
- Batch normalization calculates running statistics during training. With highly varied augmented data, these statistics may not represent the validation data well.
- **Solution:** Ensure that batch sizes are not too small to prevent noisy batch statistics. Use a batch size of at least 32 or 64.

Model tuning – lower data augmentation (model 12)

Aim – facilitate learning by reducing data augmentation complexity

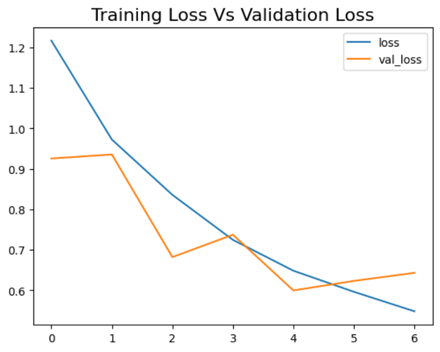


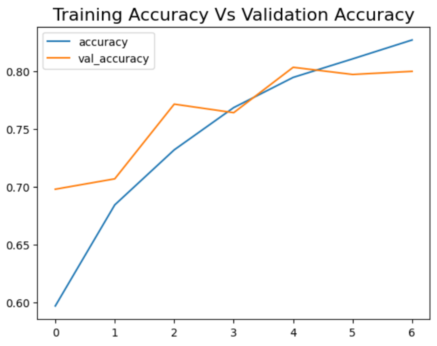
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=15,  # Reduce rotation
    width_shift_range=0.1,  # Reduce width shift
    height_shift_range=0.1,  # Reduce height shift
    shear_range=0.1,  # Reduce shear
    zoom_range=0.1,  # Reduce zoom
    horizontal_flip=True,  # Keep horizontal flip
    fill_mode='nearest'
)

datagen.fit(x_train_all)
```

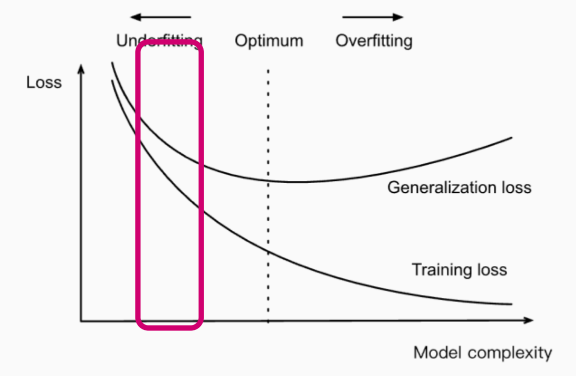


	accuracy	loss	val_accuracy	val_loss
0	0.596750	1.216773	0.6979	0.925662
1	0.684325	0.972460	0.7069	0.935512
2	0.732000	0.835807	0.7716	0.682035
3	0.768600	0.724591	0.7642	0.737224
4	0.794750	0.648303	0.8035	0.599655
5	0.810800	0.596211	0.7973	0.623190
6	0.827175	0.548088	0.8000	0.643121

Training time: 3 min 16 s



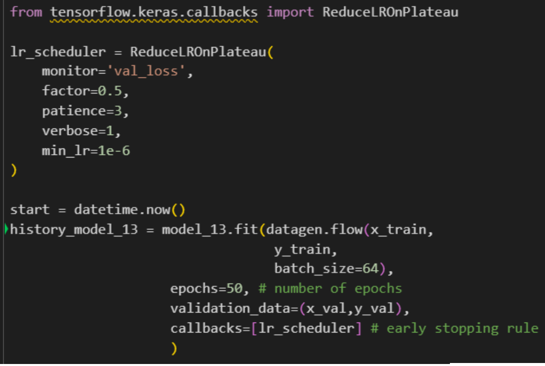




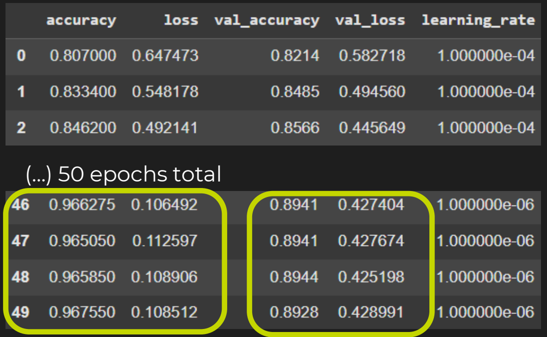
References: Kurban, 2021

Model tuning – adaptive training rate (model 13)

Aim – allow larger learning jumps to overcome higher complexity in training data (avoid local minima)



```
from tensorflow.keras.callbacks import ReduceLROnPlateau

lr_scheduler = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=3,
    verbose=1,
    min_lr=1e-6
)

start = datetime.now()
history_model_13 = model_13.fit(datagen.flow(x_train,
                                             y_train,
                                             batch_size=64),
                    epochs=50, # number of epochs
                    validation_data=(x_val,y_val),
                    callbacks=[lr_scheduler] # early stopping rule
                    )
```

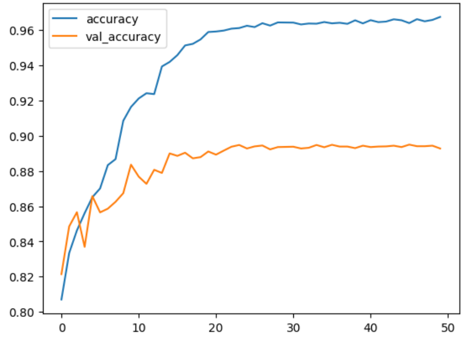


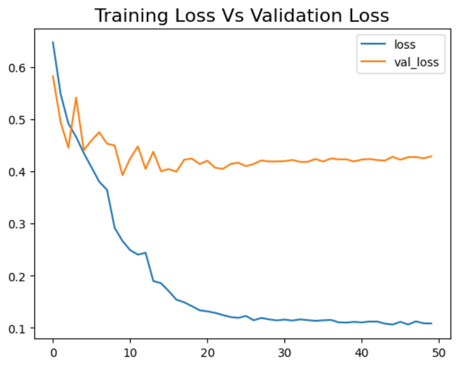
	accuracy	loss	val_accuracy	val_loss	learning_rate
0	0.807000	0.647473	0.8214	0.582718	1.000000e-04
1	0.833400	0.548178	0.8485	0.494560	1.000000e-04
2	0.846200	0.492141	0.8566	0.445649	1.000000e-04

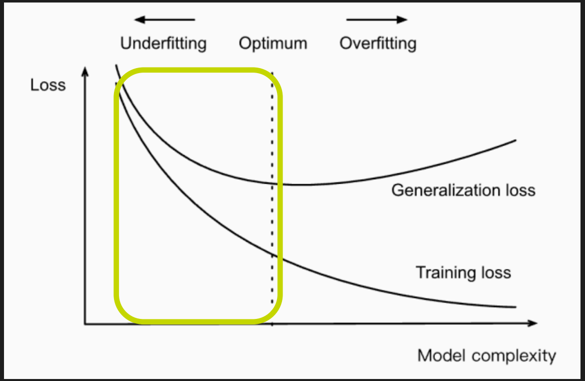
(...) 50 epochs total

46	0.966275	0.106492	0.8941	0.427404	1.000000e-06
47	0.965050	0.112597	0.8941	0.427674	1.000000e-06
48	0.965850	0.108906	0.8944	0.425198	1.000000e-06
49	0.967550	0.108512	0.8928	0.428991	1.000000e-06

Training time: 22 min 30 s







References: Kurban, 2021

Model performance

*Until early stopping rule is met or maximum training epochs reached.
Models ran on Google Colab Pro default environment and A100 GPU hardware accelerator

Model	Architecture	Optimizer	Other features	Epochs*	Time	Train accuracy	Train loss	Validation accuracy	Validation loss
1	Simple CNN	SGD	Learning rate = 1e-3	33	55 s	**0.72**	0.80	**0.64**	1.06
2	Simple CNN	Adam	Learning rate = 1e-3	9	18 s	**0.78**	0.65	**0.64**	1.15
3	**VGG-19**	SGD	Learning rate = 1e-3	5	32 s	**0.87**	0.39	**0.75**	0.86
4	VGG-19	Adam	Learning rate = 1e-3	3	30 s	**0.09**	2.30	**0.09**	2.30
5	VGG-19	Adam	Learning rate = 1e-4	3	28 s	**0.94**	0.19	**0.82**	0.66
6	VGG-19	Adam	Batch normalisation	5	44 s	**0.97**	0.09	**0.79**	1.07
7	VGG-19	Adam	Dropout	9	1 min 11 s	**0.98**	0.06	**0.82**	0.95
8	VGG-19	Adam	**Simpler dense layers**	5	41 s	**0.99**	0.03	**0.84**	0.87
9	VGG-19	Adam	**Dropout + simpler dense layers**	5	41 s	**0.99**	0.04	**0.84**	0.90
10	VGG-19	Adam	Pre-trained weights	27	1 min 17 s	**0.65**	0.99	**0.62**	1.09
11	VGG-19	Adam	Data augmentation (higher)	9	4 min 13 s	**0.58**	1.17	**0.63**	1.11
12	VGG-19	Adam	**Data augmentation (lower)**	7	3 min 16 s	**0.83**	0.54	**0.80**	0.64
13	VGG-19	Adam	**Adaptive learning rate + higher patience**	50 (max)	22 min 30 s	**0.97**	0.11	**0.89**	0.43

Critical analysis

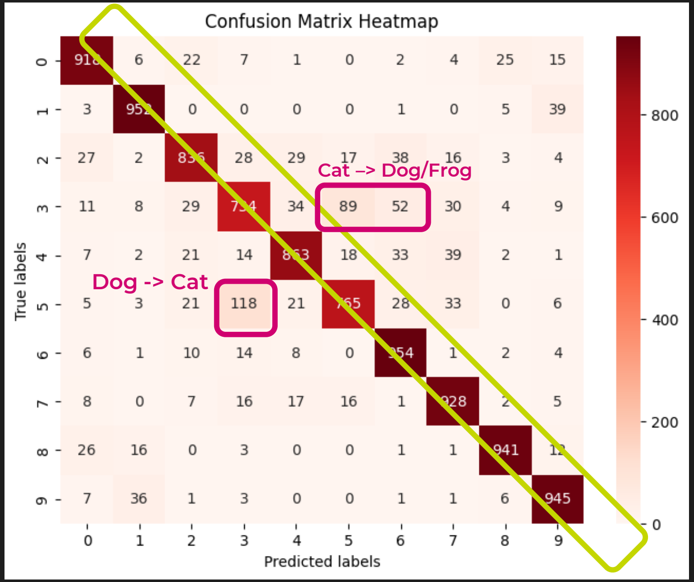
Class	Label	Precision	Recall	F1-Score	Accuracy
0	**Airplane**	**0.90**	**0.92**	**0.91**	**0.92**
1	**Automobile**	**0.93**	**0.95**	**0.94**	**0.95**
2	Bird	0.88	0.84	0.86	0.84
3	**Cat**	**0.78**	**0.73**	**0.76**	**0.73**
4	Deer	0.89	0.86	0.87	0.86
5	**Dog**	**0.85**	**0.77**	**0.80**	**0.77**
6	Frog	0.86	0.95	0.90	0.95
7	Horse	0.88	0.93	0.90	0.93
8	**Ship**	**0.95**	**0.94**	**0.95**	**0.94**
9	**Truck**	**0.91**	**0.94**	**0.93**	**0.95**
Average	N/A	0.88	0.88	0.88	0.88

```
model = model_13
```

Train accuracy: 0.97
Validation accuracy: 0.88

```
model.evaluate(x_test,y_test)
```

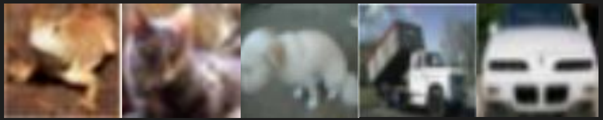
```
from sklearn.metrics import classification_report, confusion_matrix

predictions = np.argmax(model.predict(x_test), axis=-1)

print(classification_report(y_test,predictions))
```



Confusion Matrix Heatmap

True labels \ Predicted labels	0	1	2	3	4	5	6	7	8	9
0	918	6	22	7	1	0	2	4	25	15
1	3	952	0	0	0	0	1	0	5	39
2	27	2	836	28	29	17	38	16	3	4
3	11	8	29	734	34	89	52	30	4	9
4	7	2	21	14	863	18	33	39	2	1
5	5	3	21	118	21	765	28	33	0	6
6	6	1	10	14	8	0	954	1	2	4
7	8	0	7	16	17	16	1	928	2	5
8	26	16	0	3	0	0	1	1	941	12
9	7	36	1	3	0	0	1	1	6	945

```
cm=confusion_matrix(y_test,predictions)

# Plotting the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm,
            annot=True,
            fmt='g',
            cmap='Reds',
            xticklabels=np.arange(10),
            yticklabels=np.arange(10))
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix Heatmap')
plt.show()
```



CIFAR-10 sample images: frog, cat, dog, truck, car

References: Andrade, 2019; Gouk et al., 2018; Krizhevsky, 2009a;

Learning outcomes and reflections

- Importance of broad network depth (simple CNN vs VGG-19)
- Overfitting/regularisation (bias/variance) – generalisability impaired if model too complex for task
- Optimizers – faster learning vs higher training complexity
- Learning rate - computational demand vs overcoming local minima; benefits of adaptive learning rates
- Transfer learning – not necessarily useful if task/data very different (ImageNet vs CIFAR-10)
- Data augmentation – improved validation accuracy vs lower training accuracy (bias/variance), but must be careful
- Model complexity vs computing requirements – consumed 70 Google Colab computational units (free version: 10/month); many in initial debugging

Conclusion

- Explored impact of CNN architecture design and training choices on model performance
- Tackled many common machine learning challenges (e.g. vanishing gradients, overfitting, unstable learning, computing requirements)
- Final model with good performance
- Performance potentially further improved with additional hyperparameter tuning
- AI feedback loop – *AI begets AI*

References

Adhikari, T. (2023) ‘Designing a Convolutional Neural Network for Image Recognition: A Comparative Study of Different Architectures and Training Techniques’. Rochester, NY: Social Science Research Network. Available at: https://doi.org/10.2139/ssrn.4366645.

Andrade, A. de (2019) ‘Best Practices for Convolutional Neural Networks Applied to Object Recognition in Images’. arXiv. Available at: https://doi.org/10.48550/arXiv.1910.13029.

Augmented AI (2023) How to Implement Object Detection Using Deep Learning: A Step-by-Step Guide. Available at: https://www.augmentedstartups.com/blog/how-to-implement-object-detection-using-deep-learning-a-step-by-step-guide.

Bag, S. (2023) ‘Activation Functions — All You Need To Know!’, Analytics Vidhya, 2 January. Available at: https://medium.com/analytics-vidhya/activation-functions-all-you-need-to-know-355a850d025e (Accessed: 20 January 2025).

Belagatti, P. (2024) Understanding the Softmax Activation Function: A Comprehensive Guide, SingleStore. Available at: https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/ (Accessed: 20 January 2025).

Berrar, D. (2019) ‘Cross-Validation’, in Encyclopedia of Bioinformatics and Computational Biology, pp. 542–545. Available at: https://www.sciencedirect.com/topics/medicine-and-dentistry/cross-validation (Accessed: 20 January 2025).

Bronshtein, A. (2020) Train/Test Split and Cross Validation in Python, Medium. Available at: https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6 (Accessed: 17 January 2025).

Chang, D. and Pathak, A. (2020) Effect of Batch Size on Neural Net Training | by Daryl Chang | Deep Learning Experiments | Medium. Available at: https://medium.com/deep-learning-experiments/effect-of-batch-size-on-neural-net-training-c5ae8516e57 (Accessed: 20 January 2025).

Chang, P.D. et al. (2018) ‘Hybrid 3D/2D Convolutional Neural Network for Hemorrhage Evaluation on Head CT’, American Journal of Neuroradiology, 39(9), pp. 1609–1616. Available at: https://doi.org/10.3174/ajnr.A5742.

Fortuner, B. (2017) Loss Functions — ML Glossary documentation. Available at: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html (Accessed: 20 January 2025).

Giordano, D. (2020) 7 tips to choose the best optimizer, Medium. Available at: https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e (Accessed: 20 January 2025).

Gouk, H. et al. (2018) ‘MaxGain: Regularisation of Neural Networks by Constraining Activation Magnitudes’. arXiv. Available at: https://doi.org/10.48550/arXiv.1804.05965.

Hughes, C. (2024) ‘A Brief Overview of Cross Entropy Loss’, Medium, 25 September. Available at: https://medium.com/@chris.p.hughes10/a-brief-overview-of-cross-entropy-loss-523aa56b75d5 (Accessed: 20 January 2025).

Jiang, L. (2020) A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp, Adam), Medium. Available at: https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c (Accessed: 20 January 2025).

Keras: Deep Learning for humans (no date). Available at: https://keras.io/ (Accessed: 17 January 2025).

Kingma, D.P. and Ba, J. (2017) ‘Adam: A Method for Stochastic Optimization’. arXiv. Available at: https://doi.org/10.48550/arXiv.1412.6980.

Krizhevsky, A. (2009a) CIFAR-10 and CIFAR-100 datasets. Available at: https://www.cs.toronto.edu/~kriz/cifar.html (Accessed: 17 January 2025).

Krizhevsky, A. (2009b) ‘Learning Multiple Layers of Features from Tiny Images’, in. Available at: https://api.semanticscholar.org/CorpusID:18268744.

Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2012) ‘ImageNet Classification with Deep Convolutional Neural Networks’, in Advances in Neural Information Processing Systems. Curran Associates, Inc. Available at: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (Accessed: 20 January 2025).

Kurban, R. (2021) Boost your Network Performance, Medium. Available at: https://towardsdatascience.com/boost-your-network-performance-cc0a2a95c5ef (Accessed: 20 January 2025).

Lu, S.-A. (2017) ‘SGD > Adam?? Which One Is The Best Optimizer: Dogs-VS-Cats Toy Experiment’, SALu, 28 May. Available at: https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/ (Accessed: 20 January 2025).

Musstafa (2022) ‘Optimizers in Deep Learning’, Medium, 12 February. Available at: https://musstafa0804.medium.com/optimizers-in-deep-learning-7bf81fed78a0 (Accessed: 20 January 2025).

Natras, R., Soja, B. and Schmidt, M. (2022) ‘Ensemble Machine Learning of Random Forest, AdaBoost and XGBoost for Vertical Total Electron Content Forecasting’, Remote Sensing, 14(15), p. 3547. Available at: https://doi.org/10.3390/rs14153547.

Sarki, R. et al. (2022) ‘Automated detection of COVID-19 through convolutional neural network using chest x-ray images’, PLOS ONE, 17(1), p. e0262052. Available at: https://doi.org/10.1371/journal.pone.0262052.

Simonyan, K. and Zisserman, A. (2015) ‘Very Deep Convolutional Networks for Large-Scale Image Recognition’. arXiv. Available at: https://doi.org/10.48550/arXiv.1409.1556.

Solawetz, J. (2021) An Introduction to ImageNet, Roboflow Blog. Available at: https://blog.roboflow.com/introduction-to-imagenet/ (Accessed: 20 January 2025).

Srivastava, S. (2024) ‘Understanding the Difference Between ReLU and Sigmoid Activation Functions in Deep Learning’, Medium, 18 April. Available at: https://medium.com/@srivastavashivansh8922/understanding-the-difference-between-relu-and-sigmoid-activation-functions-in-deep-learning-33b280fc2071 (Accessed: 20 January 2025).

Stanford Vision Lab, Stanford University, Princeton University (2021) ImageNet. Available at: https://image-net.org/index.php (Accessed: 20 January 2025).

Swapna. (2020) Convolutional Neural Network | Deep Learning, Developers Breach. Available at: https://developersbreach.com/convolution-neural-network-deep-learning/ (Accessed: 20 January 2025).

ubiAI (2023) Introduction to the COCO Dataset. Available at: https://ubiai.tools/introduction-to-the-coco-dataset/.

Wilson, A.C. et al. (2018) ‘The Marginal Value of Adaptive Gradient Methods in Machine Learning’. arXiv. Available at: https://doi.org/10.48550/arXiv.1705.08292.

Zulkifli, H. (2018) Understanding Learning Rates and How It Improves Performance in Deep Learning, Medium. Available at: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10 (Accessed: 20 January 2025).

Neural Network Models for Object Recognition

Postgraduate Diploma in Artificial Intelligence
Machine Learning module

Guilherme Amorim

Jan 2025

University of Essex