Convolutional Neural Networks 👀

By:

Anne Valvezan
Manish Kumar Pandey
Eric Salsac

Technical Overview

What are the layers?

Convolutional layer - (Receptive Field Layer - *uses a scanning grid*)

- Takes a tensor of form:
  - [Batch Number , Width , Height , Colour (RGB)]

- Outputs a tensor of form:
  - [Batch Number , Width , Height , Number of Feature Maps]

    For example it can take an input of type (24, 24, 3) and the filter count is 32 then the output will be of size (24, 24, 32). So we can say that we have our image with 32 feature maps now. Things to be careful at the convolution layer are the kernel size and padding. Because with “valid” padding we can have reduced size as the output.

The feature maps, instead of telling us colour values at a given position, give us feature intensity at a given position.

(Filter (convolutional kernel) count is what determines the number of feature maps you have.)

Pooling Layer - (Receptive Field Layer - *uses a scanning grid*)

- Subsamples the image by:
  - Max
  - Avg
- Reduces:
  - Computational load
  - Memory usage
  - Number of parameters, to (attempt to) reduce overfitting.

This is the layer which is responsible for the downsampling of the image. The key parameters to take care of are the pool size and stride. Pool size tells us how many pixels at a time we are going to downsample and stride tells us the how this grid will move, with overlap or without. For example a pool size (2,2) and stride (2, 2) will downsample the data by 75%. The Max pooling and Average pooling are like intensifying the pixels vs blurring the pixels.

Batch Normalisation Layer (before or after activation layer)

- Normalises the outputs of a convolution-activation cycle for a batch
- Scales and offsets the output for that batch

This is done to try and prevent gradients from exploding/vanishing while not destroying the information by simply normalising everything.

Activation Layer

This is the Layer where we add our non-linearity to the model by calling the activation function.

We use mostly Relu as our activation function.

Flatten(ing) Layer

After getting all our informations from the Convolution layers we turn our information into the vectors for the final classification.

Turns our tensors with our feature maps into a vector so our dense neural layer can understand it.

Dense Layer

Typical neural and activation layers to make sense of the flattened vector produced by the convolutional layers and to perform classification or regression in the final layers.

Simple Convolutional Neural Network Architecture

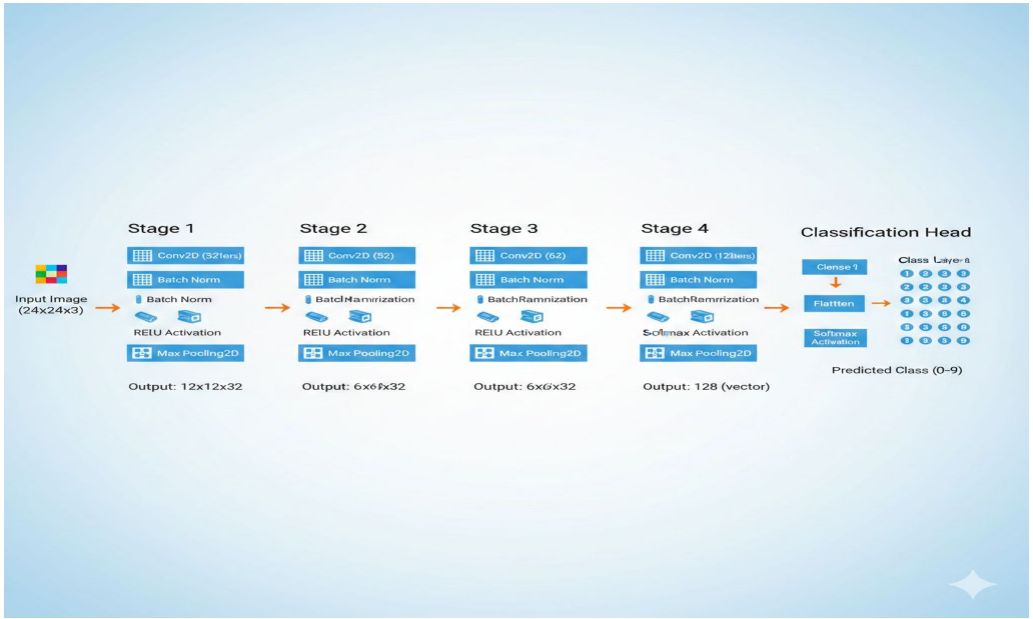
After defining tensor (image) size:

- **C**onvolutional layer of small filter (convolution kernel) count
- **B**atch normalisation
- **A**ctivation layer
- **C**onvolutional layer of small filter (convolution kernel) count
- **B**atch normalisation
- **A**ctivation layer
- **P**ooling layer

Repeat the above points for larger filter sizes then:

- Flatten
- 1-2 Dense normal neural layers where the final is used for classification or regression as needed

A rough image of the model architecture



Data Processing

The Prep-work

Loading Dataset and Dataset Overview

Cifar10 image dataset

- 60,000 images
- 32 by 32 pixels with 3 channel RGB colour
- 10 classes of image
- Accessible from within tensorflow.keras
- Test-train split already done:

  (10,000 test/validation images, 50,000 training images)

Train-Test-Valid Split

- Training data
  - To train the model
- Validation data
  - To allow the model to self evaluate and adjust its own parameters
- Test data
  - For a final evaluation of the model that the model has not seen (data leakage is bad)

Normalising Data

- Treat data as float and divide by 256, our largest dimension (colour)

- Not the same as batch normalisation, they perform different roles. It is standard to use both.

Y’s Transformation to Categorical

- Simply a function where you state the number of classes:

  y_train = to_categorical(y_train, num_classes=10)

- Or just use sparse_categorical_crossentropy instead of categorical_crossentropy

  The only difference is the expected formatting of labels.

Building an Image Augmentor - ‘Deprecated’ version

```
idg = ImageDataGenerator(
    rotation_range=5,                    # random rotation up to 15 degrees
    width_shift_range=0.05,              # horizontal shift
    height_shift_range=0.05,             # vertical shift
    horizontal_flip=True,                # randomly flip images
    zoom_range=0.1,                      # zoom in/out
    fill_mode="nearest",                 # fill missing pixels with nearest values
    brightness_range = [0.9, 1.1])       # change image “brightness”


idg.fit(X_train)                         # Turns out not to be necessary for our uses
image_augmentor = idg.flow(  X_train, y_train, batch_size=64)        # Can be memory intensive
```

Image Augmentation - The Better Supported Way

Write your augmentations as a sequential model using layers and add them to the start of you model.

Set your batch size via .fit()

Done.

‘Transfer’ Learning

A Generous Title

Transfer Learning - Frankenstein’s Rubber-boned CNN

The idea is simply to:

- Take a model
- Remove its “top layer” - (the dense one)
- Supply your own
- Play with hyper-parameters
- Unfreeze layers so parts of the model are able to learn from your data.

Models run:

- MobileNet V2
- Resnet50 V2
- ConvNextTiny
- NASNet
- DenseNet169

Image Scaling - The Biggest Hurdle

Resolutions used:

- 32,32,3
- 64,64,3
- 96,96,3
- Anne’s 224 pixel juggernaut

Metrics: ConvNextTiny

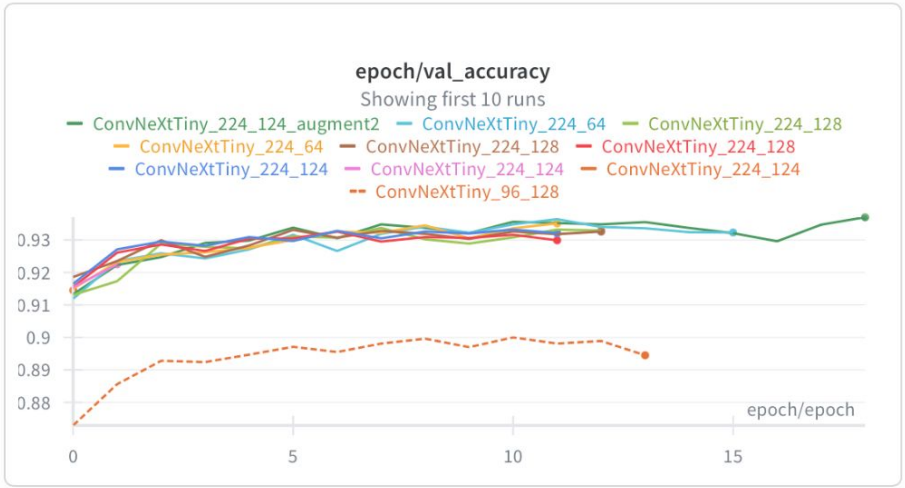
MODEL_NAME	INPUT_SHAPI	BATCH_SIZ	AUGMENT	FINAL_VAL_ACCURAC	FINAL_VAL_LOS	RUNTIME	EPOCHS	LEARNING_RATE	EPOCH/ACCURACY	EPOCH/LOSS	EPOCH/VAL_ACCURACY	EPOCH/VAL_LOSS
ConvNeXtTiny	[224,224,3]	124	true	0.9263	0.22109	37m 7s	20	0.001	0.95557	0.12695	0.937	0.20135
ConvNeXtTiny	[224,224,3]	64	true	0.9261	0.22527	30m 21s	20	0.001	0.95407	0.12833	0.9323	0.23347
ConvNeXtTiny	[224,224,3]	128	true	0.9254	0.22334	25m 2s	20	0.001	0.94108	0.16449	0.9329	0.2017
ConvNeXtTiny	[224,224,3]	64	true	0.924	0.22211	22m 58s	20	0.001	0.9487	0.14898	0.9351	0.20065
ConvNeXtTiny	[224,224,3]	128	true	0.9223	0.22731	25m 2s	20	0.001	0.94103	0.16622	0.9326	0.20686
ConvNeXtTiny	[224,224,3]	128	false	0.9221	0.2359	23m 4s	20	0.001	0.9811	0.065911	0.9299	0.23092
ConvNeXtTiny	[224,224,3]	124	-	0.9204	0.23975	23m 43s	20	0.001	0.97855	0.07019	0.9321	0.22685
ConvNeXtTiny	[224,224,3]	124	true	0.9144	0.25129	5m 34s	2	0.001	0.90317	0.28744	0.9225	0.22825
ConvNeXtTiny	[224,224,3]	124	true	0.9099	0.27492	2m 28s	1	0.001	0.85212	0.46852	0.9145	0.25738
ConvNeXtTiny	[96,96,3]	128	true	0.8921	0.32865	5m 2s	20	0.001	0.93888	0.17441	0.8945	0.32966
ConvNeXtTiny	[96,96,3]	128	true	0.8899	0.32641	4m 23s	20	0.001	0.93198	0.19388	0.8977	0.31882
ConvNeXtTiny	[224,224,3]	124	true	0.8859	0.32862	39m 5s	20	0.001	0.91215	0.24851	0.8932	0.31131
ConvNeXtTiny	[96,96,3]	64	true	0.8852	0.34322	3m 18s	20	0.001	0.92125	0.22115	0.8918	0.33192
ConvNeXtTiny	[96,96,3]	64	true	0.8843	0.33804	3m 15s	20	0.001	0.92255	0.22005	0.8927	0.32061

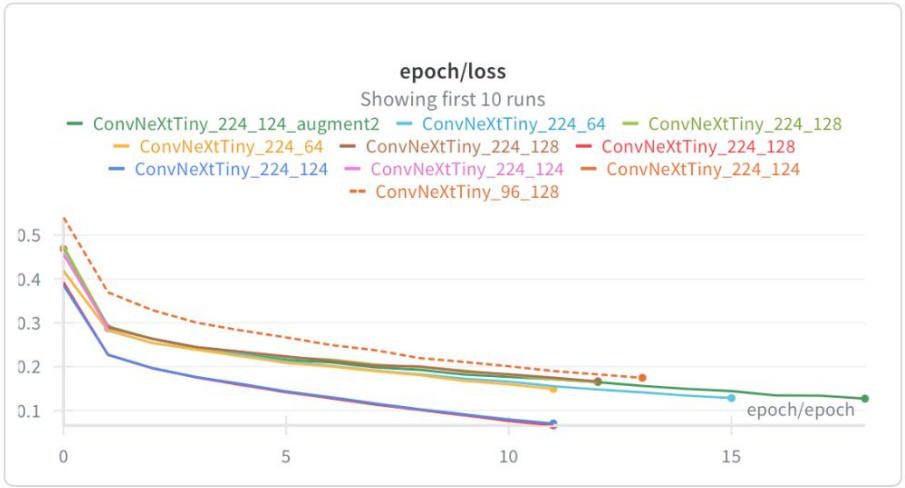
Metrics: MobileNetV2

MODEL_NAME	INPUT_SHAPI	BATCH_SIZ	AUGMENT	FINAL_VAL_ACCURAC	FINAL_VAL_LOS	RUNTIME	EPOCHS	LEARNING_RATE	EPOCH/ACCURACY	EPOCH/LOSS	EPOCH/VAL_ACCURACY	EPOCH/VAL_LOSS
MobileNetV2	[96,96,3]	128	true	0.8587	0.41476	1m 20s	20	0.001	0.8905	0.31069	0.8623	0.4377
MobileNetV2	[224,224,3]	128	true	0.857	0.4182	7m 30s	20	0.001	0.88608	0.32031	0.8619	0.40815
MobileNetV2	[96,96,3]	64	true	0.8558	0.41997	57s	20	0.001	0.86555	0.37557	0.856	0.43426
MobileNetV2	[224,224,3]	64	true	0.8543	0.44929	8m 5s	20	0.001	0.89318	0.30089	0.8632	0.41078
MobileNetV2	[224,224,3]	128	true	0.8543	0.42503	7m 52s	20	0.001	0.8832	0.3332	0.8586	0.42352
MobileNetV2	[96,96,3]	128	true	0.8539	0.42679	1m 6s	20	0.001	0.87963	0.34032	0.8585	0.4351

Better models are better ! Or are they better optimised?

Curves


epoch/val_accuracy
Showing first 10 runs
ConvNeXtTiny_224_124_augment2
ConvNeXtTiny_224_64
ConvNeXtTiny_224_128
ConvNeXtTiny_224_64
ConvNeXtTiny_224_128
ConvNeXtTiny_224_128
ConvNeXtTiny_224_124
ConvNeXtTiny_224_124
ConvNeXtTiny_224_124
ConvNeXtTiny_96_128
0.93
0.92
0.91
0.9
0.89
0.88
epoch/epoch
0
5
10
15


epoch/loss
Showing first 10 runs
ConvNeXtTiny_224_124_augment2
ConvNeXtTiny_224_64
ConvNeXtTiny_224_128
ConvNeXtTiny_224_64
ConvNeXtTiny_224_128
ConvNeXtTiny_224_128
ConvNeXtTiny_224_124
ConvNeXtTiny_224_124
ConvNeXtTiny_224_124
ConvNeXtTiny_96_128
0.5
0.4
0.3
0.2
0.1
epoch/epoch
0
5
10
15

Observed Patterns

Layer Unfreezing - Scale and Position

Model Name	Accuracy	Precision	Recall	F1_Score
MNv2 Frozen scaled to 64,64,3	0.6799	0.678568	0.6799	0.678798
MNv2 top 40 Unfrozen scaled to 64,64,3	0.7069	0.770010	0.7069	0.698364
MNv2 (-80:-40) Unfrozen scaled to 64,64,3	0.8608	0.863642	0.8608	0.859554
MNv2 (-120:-80) Unfrozen scaled to 64,64,3	0.8318	0.838674	0.8318	0.825733
MNv2 (0:40) Unfrozen scaled to 64,64,3	0.8162	0.830133	0.8162	0.812254

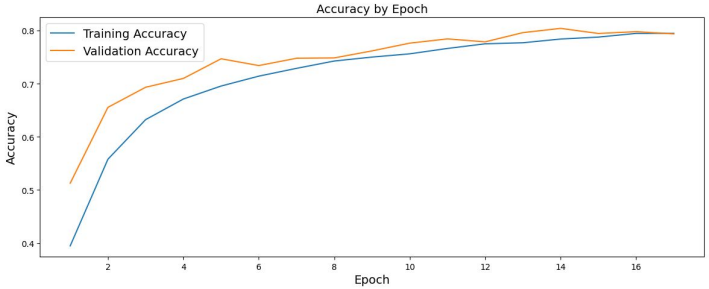
Unfreezing position seems to matter.

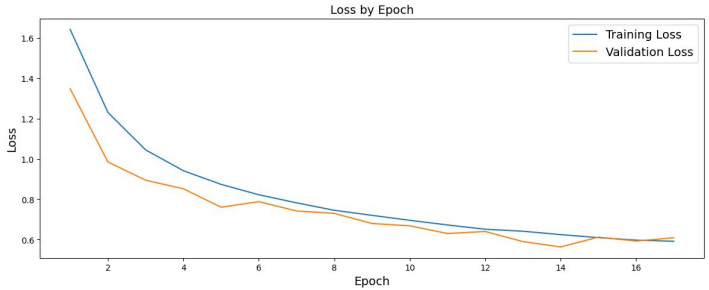
However resource and time limitations prevented the scaling of MNv2 (-80:-40) to 96,96,3, potentially being our best performer.

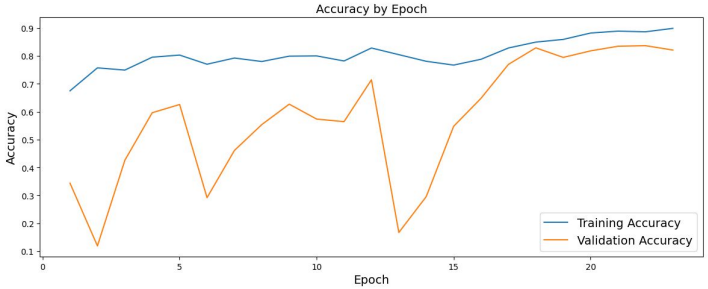
Epoch graphs and Patience (Early Stopping)

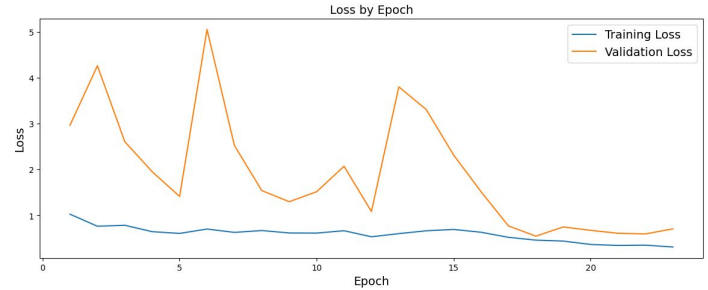
- Stable learning rates need lower patience values for early stopping.
- Unstable learning rates require higher values, usually models that have layers unfrozen.

Stable learning









Unstable learning

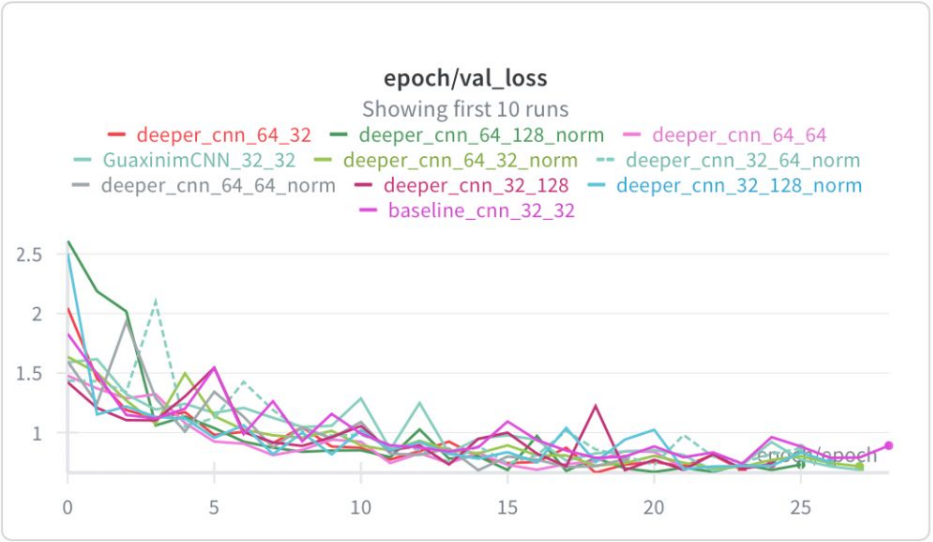
Guaxinim Cnn (Raccoon CNN)

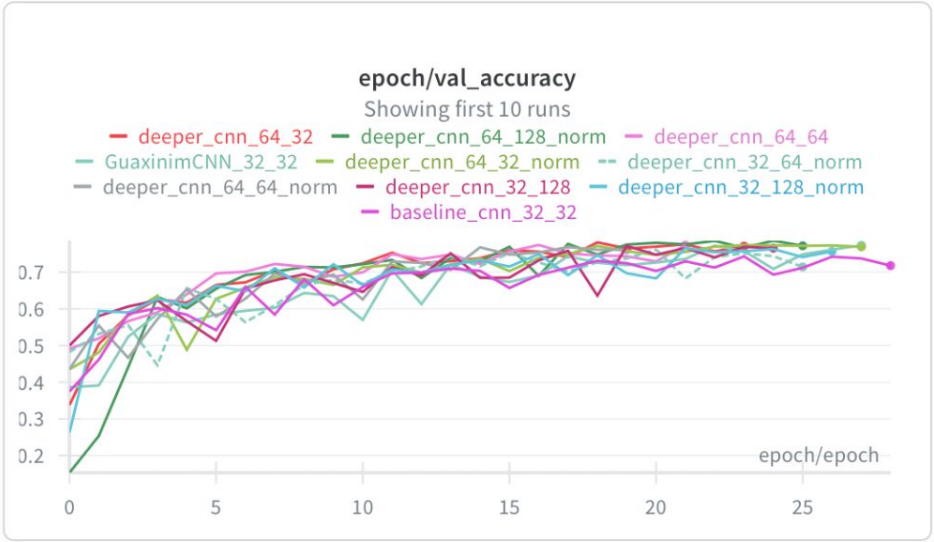
MODEL_NAME	INPUT_SHAPE	BATCH_SIZE	AUGMENT	FINAL_VAL_A	FINAL_VAL_LOS	RUNTIME	EPOCHS	LEARNING_RATE	NORMALIZE	EPOCH/VAL_ACC	EPOCH/VAL_LOS
GuaxinimCNN	[32,32,3]	32	true	0.7632	0.71233	2m 49s	30	0.001	false	0.773	0.68804
GuaxinimCNN	[32,32,3]	64	true	0.7115	0.87548	1m 13s	30	0.001	false	0.7153	0.85284
GuaxinimCNN	[32,32,3]	32	true	0.7032	0.89318	3m 25s	20	0.001	false	0.7091	0.87517
GuaxinimCNN	[64,64,3]	64	true	0.6843	0.97136	2m 51s	30	0.001	false	0.6854	0.94577
GuaxinimCNN	[32,32,3]	32	true	0.6314	1.16058	2m 15s	30	0.001	true	0.6517	1.13443
GuaxinimCNN	[32,32,3]	128	true	0.2514	1.84612	28s	30	0.001	false	0.2654	1.85026
GuaxinimCNN	[32,32,3]	64	true	0.2365	2.1278	22s	30	0.001	true	0.2507	2.88495
GuaxinimCNN	[32,32,3]	128	true	0.23	2.0492	17s	30	0.001	true	0.2345	3.24194
GuaxinimCNN	[64,64,3]	128	true	0.1	2.3026	39s	30	0.001	false	0.0956	2.3027
GuaxinimCNN	[64,64,3]	32	true	0.1	2.30261	1m 58s	30	0.001	false	0.0956	2.30271

Deeper Cnn

MODEL_NAME	INPUT_SHAPE	BATCH_SIZE	AUGMENT	FINAL_VAL_A	FINAL_VAL_LOS	RUNTIME	EPOCHS	LEARNING_RATE	NORMALIZE	EPOCH/VAL_ACC	EPOCH/VAL_LOS
deeper_cnn	[64,64,3]	32	true	0.7745	0.68709	1m 37s	30	0.001	false	0.7708	0.68768
deeper_cnn	[64,64,3]	128	true	0.7696	0.72646	1m 9s	30	0.001	true	0.7717	0.73134
deeper_cnn	[64,64,3]	64	true	0.7682	0.71935	1m 7s	30	0.001	false	0.7735	0.7042
deeper_cnn	[64,64,3]	32	true	0.7627	0.72995	1m 55s	30	0.001	true	0.769	0.71666
deeper_cnn	[32,32,3]	64	true	0.7587	0.73761	47s	30	0.001	true	0.7143	0.86841
deeper_cnn	[64,64,3]	64	true	0.7565	0.72698	1m 1s	30	0.001	true	0.7613	0.77778
deeper_cnn	[32,32,3]	128	true	0.7548	0.73076	30s	30	0.001	false	0.7653	0.73827
deeper_cnn	[32,32,3]	128	true	0.7499	0.74118	33s	30	0.001	true	0.7565	0.75058
deeper_cnn	[32,32,3]	32	true	0.7404	0.77271	1m 8s	30	0.001	true	0.7582	0.74004
deeper_cnn	[64,64,3]	128	true	0.7372	0.76484	48s	30	0.001	false	0.7317	0.90243

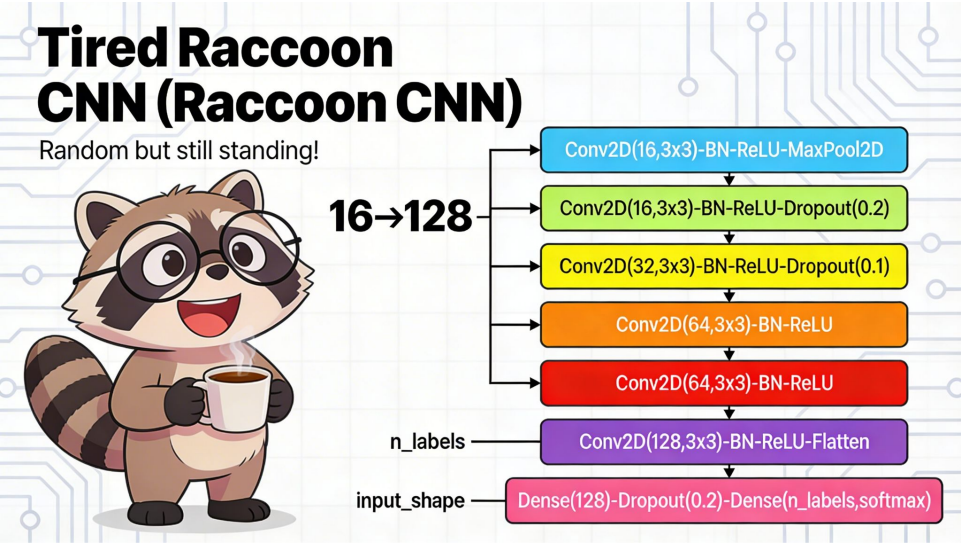

Curves Slide
Watch the Magic Happen!
Accuracy
Loss

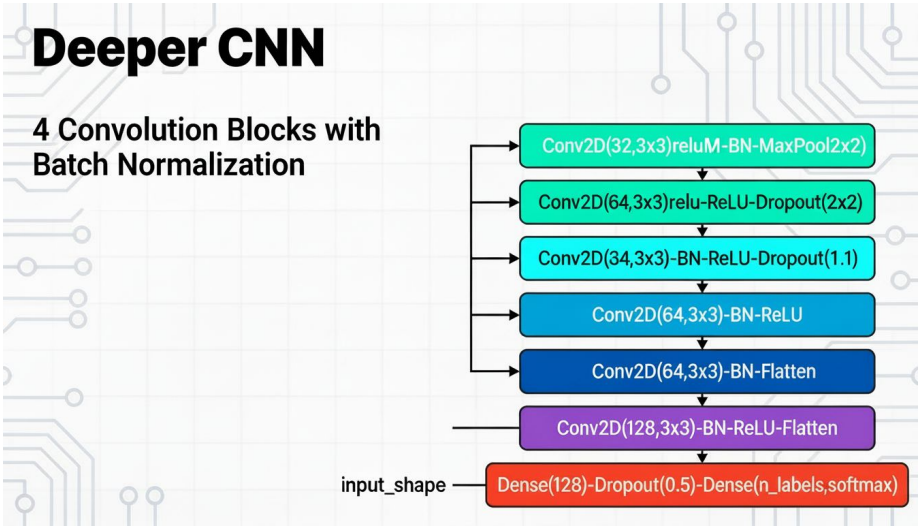

epoch/val_loss
Showing first 10 runs
deeper_cnn_64_32
deeper_cnn_64_128_norm
deeper_cnn_64_64
GuaxinimCNN_32_32
deeper_cnn_64_32_norm
deeper_cnn_32_64_norm
deeper_cnn_64_64_norm
deeper_cnn_32_128
deeper_cnn_32_128_norm
baseline_cnn_32_32
2.5
2
1.5
1
0
5
10
15
20
25
epoch/epoch


epoch/val_accuracy
Showing first 10 runs
deeper_cnn_64_32
deeper_cnn_64_128_norm
deeper_cnn_64_64
GuaxinimCNN_32_32
deeper_cnn_64_32_norm
deeper_cnn_32_64_norm
deeper_cnn_64_64_norm
deeper_cnn_32_128
deeper_cnn_32_128_norm
baseline_cnn_32_32
0.7
0.6
0.5
0.4
0.3
0.2
0
5
10
15
20
25
epoch/epoch

Conclusions

- If you use Pooling you need bigger input images sizes
- It is possible to achieve similar results with dropout and minimal pooling
- Normalization not always helps (at least for this size of images)





Original CNN vs Transfer Learning

Small Image, Big Cost

Models - Local CNN vs MNv2 vs ResNv2

MNv2 at 96,96,3 is technically the best model of **this** set of tests.

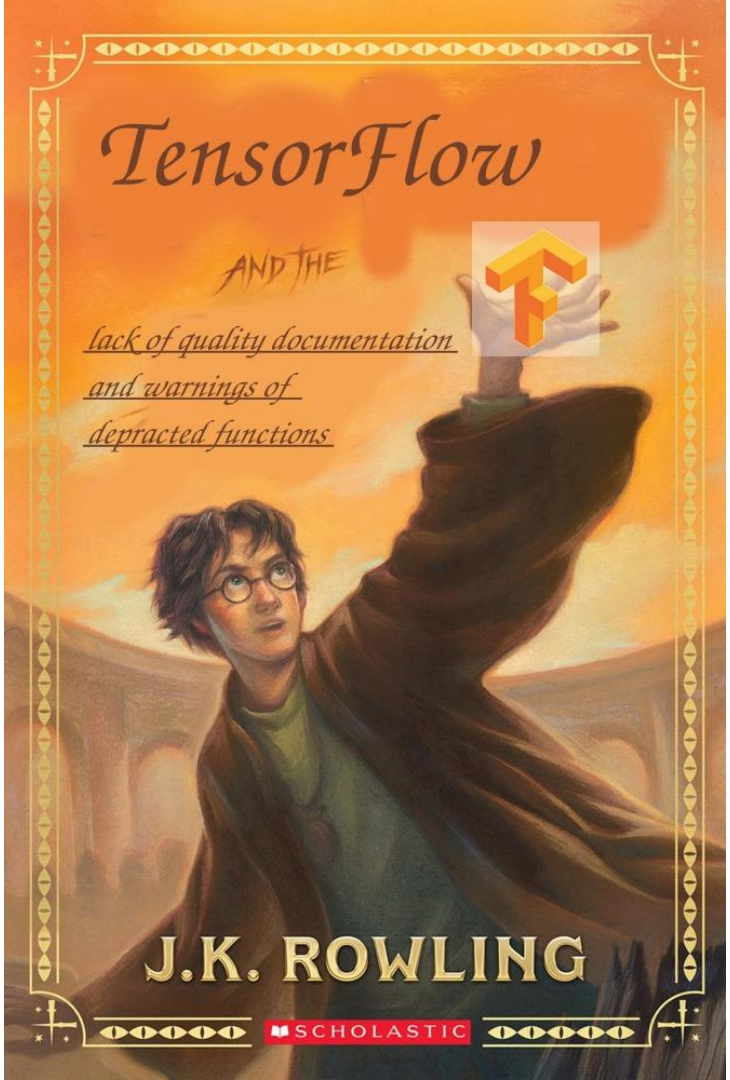
However the computational cost of running these models at higher resolution does not appear to be worth it.

Local CNN2 is running images at 32,32,3 and achieving 80%. Scaling this solution seems to be the more efficient option for this image set.

Local CNN2 also does not benefit from the scaling.

Model Name	Accuracy	Precision	Recall	F1_Score
Local Convolutional Nueral Network 1	0.6327	0.632851	0.6327	0.632243
Local Convolutional Nueral Network 2	0.8049	0.804564	0.8049	0.801050
MNv2 Frozen scaled to 64,64,3	0.6799	0.678568	0.6799	0.678798
MNv2 top 40 Unfrozen scaled to 64,64,3	0.7069	0.770010	0.7069	0.698364
MNv2 top 20 Unfrozen scaled to 64,64,3	0.8424	0.856747	0.8424	0.842680
ResN50v2 Frozen scaled to 64,64,3	0.6748	0.672565	0.6748	0.672547
ResN50v2 top 49 Unfrozen scaled to 64,64,3	0.8191	0.828033	0.8191	0.817325
MNv2 top 40 Unfrozen scaled to 96,96,3	0.9005	0.904373	0.9005	0.900540
ResN50v2 top 49 Unfrozen scaled to 96,96,3	0.8496	0.850810	0.8496	0.849108
MNv2 (-80:-40) Unfrozen scaled to 64,64,3	0.8608	0.863642	0.8608	0.859554
MNv2 (-120:-80) Unfrozen scaled to 64,64,3	0.8318	0.838674	0.8318	0.825733
MNv2 (0:40) Unfrozen scaled to 64,64,3	0.8162	0.830133	0.8162	0.812254
Local Convolutional Nueral Network 2 scaled to...	0.7993	0.799280	0.7993	0.797237

General Overview of Models and Closing Thoughts


TensorFlow
AND THE
lack of quality documentation
and warnings of
depracted functions
J.K. ROWLING
SCHOLASTIC