COMP4107 Final Project Report

Classifying Pokemon Types From Images Using a Convolutional Neural Network (CNN)

An Ha (101034784)
Abrar Kazi (101038318)

Introduction

In the Pokemon franchise, there are a total of 18 unique types of Pokemon, which refer to their elemental property. Moreover, for each Pokemon, they have a primary type, and optionally, a secondary type (“Types,” n.d.). Additionally, in the first generation of Pokemon, there are a total of 151 Pokemon with 15 unique types ("Generation I", 2021). Our dataset contains 150 of these Pokemon (Zhang, 2020). Thus, this project aims to classify images of these 150 Pokemon into the 15 unique primary types using a Convolutional Neural Network (CNN). CNNs are generally more powerful than a traditional Feed Forward Neural Network (FFNN), especially for image classification tasks due to the reduction of parameters when training the network, which is why we chose them for our problem. In addition, we would like to investigate the techniques and parameters that affect a CNN’s performance. These include data augmentation, image resolution, batch size, and model architecture.

Related Works

We will be modifying existing CNN model architectures investigated in previous assignments, namely, the 5th assignment in the Neural Networks course at Carleton University on the CIFAR-10 dataset, as well as investigating new model architectures. The CIFAR-10 dataset is related since it is an image classification problem, and we have explored in the 5th assignment the performance of a CNN on this dataset, achieving about 70% accuracy. We will be going a step further and investigate and apply the techniques used in achieving about 90% accuracy on the CIFAR-10 dataset to our Pokemon dataset (Kumar, 2018).

Data

The Pokemon image dataset that we used is from Kaggle, titled “7,000 Labeled Pokemon,” provided by Lance Zhang, and contains exactly 6,837 images of Pokemon from generation 1. These images are mainly an assortment of computer animated and hand drawn images, but also a few images of action figures or plushies. There are approximately 25 to 50 images for each Pokemon with each one being centered. Moreover, these images are mostly in

the JPEG format, but some images are in the PNG, and SVG formats as well (Zhang, 2020). We used a script utilizing the ImageMagick tool to convert all the images to the JPEG format ("Inline Image Modification", n.d.). Unfortunately, 8 of these images failed to convert to JPEG using the ImageMagick tool. Thus, a total of 6,829 images were used for our dataset. Additionally, these images vary in resolution (Zhang, 2020), but we used TensorFlow to resize the images to a fixed size before training our network ("tf.image.resize", 2021). Furthermore, these images are organized by the Pokemon's name, not the Pokemon's primary type (Zhang, 2020). Thus, a CSV file from Kaggle, titled "Pokemon Image Dataset," provided by Vishal Subbiah, in addition to a script parsing this CSV file, was used to sort these images into their correct folders organized by type. This CSV file contains all the Pokemon from generation 1 to 7, along with their primary and secondary type, where the Pokemon's name is listed in the first column, primary type listed in the second column, and secondary type listed in the third column. Since we were only concerned with classifying Pokemon based on their primary type, we did not use the information in the third column, which stores the Pokemon's secondary type. In addition to the CSV file provided by Vishal Subbiah, they included a dataset where it contains 1 image for each Pokemon. However, we did not add this to our dataset for training since there is only 1 image for each Pokemon (Subbiah, 2019). Thus, after converting every image to JPEG, organizing each Pokemon image into a Pokemon types folder, we train our networks on this dataset of 6,829 images with a 70-15-15 training, validation, and test split. Additionally, since the dataset is quite small in comparison to datasets like CIFAR-10 with 60,000 total images, we apply techniques such as data augmentation to boost the dataset size and normalize the images by dividing by 255 to obtain a mean close to 0 which speeds up training without impacting performance (Stottner, 2019).

Methodology

We analyzed several components of CNN architecture and training parameters to design

an optimal model. First, we investigated data augmentation, a technique for transforming input images in realistic ways. To test the effects of data augmentation, we created the following base model:

```
Model: "base_model"
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
batch_normalization (BatchNo (None, 32, 32, 32)        128
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
conv2d_1 (Conv2D)            (None, 16, 16, 64)        18496
batch_normalization_1 (Batch (None, 16, 16, 64)        256
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
conv2d_2 (Conv2D)            (None, 8, 8, 128)         73856
batch_normalization_2 (Batch (None, 8, 8, 128)         512
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 128)         0
flatten (Flatten)            (None, 2048)              0
dense (Dense)                (None, 128)               262272
dropout (Dropout)            (None, 128)               0
dense_1 (Dense)              (None, 15)                1935
```

Each Conv2D layer has a window size of 3x3, padding enabled, and ReLU activation function. We chose a window size of 3x3 because it results in less computation as well as using fewer parameters compared to a window size like 7x7. Furthermore, a stack of three Conv2D layers with 3x3 windows is much more powerful compared to a single Conv2D layer with a 7x7 window, since as we go down the layers, the deeper layer windows scan patches that are effectively larger. That is, the first layer 3x3 window scans a patch of size 3x3, but the second layer 3x3 window effectively scans a patch of size 5x5. Similarly, the third layer 3x3 window effectively scans a patch of 7x7. Moreover, a 3 layer stack of 3x3 windows has about 45% reduction in the total number of parameters compared to a single layer with a 7x7 window. Not only that, but we know that deep networks generally perform better than wide networks. Also, we have padding enabled on the Conv2D layers because we want the output size to not change before it goes to the max pooling layer. Otherwise, we would be losing information around the edges. Taking the important features and reducing the size of the feature maps should be the max pooling layer's job, not the Conv2D layer. In addition, padding in the Conv2D layer is often recommended in modern CNN design. We also used the ReLU activation function over the alternatives because it is a common activation function that generally performs well. For

example, in 2012, AlexNet had a similar architecture to LeNet5 but replaced the tanh activation function with the ReLU activation function, among other small architectural changes, and outperformed many other architectures that were not using the ReLU activation function (Varma & Das, n.d.). Additionally, in the base model, the 3 Conv2D layers have filters of size 32, 64, and 128 respectively because we expect that the first layer will learn primitive features such as lines and edges, needing only 32 filters. As we go deeper in the convolutional layers, it will be able to recognize more abstract features such as eyes, nose and hopefully all the Pokemon out of the 150 that we are training the network on, hence needing more filters, which are 64 and 128 respectively in the 2nd and 3rd layer.

Moreover, we added max pooling layers with a window size of 2x2, stride of 2, and no padding after each convolutional layer to considerably reduce the number of parameters needed to train in the network. These values were chosen because they are the most common for max pooling and that padding is not usually done for pooling layers (Varma & Das, n.d.). Furthermore, after each convolutional layer, we apply a batch normalization which allows us to speed up the training with fewer steps without affecting the performance of the model as well as prevent overfitting (Kumar, 2018). Finally, after the convolutional and max pooling layers, we added a fully connected dense layer with 128 neurons, ReLU activation, and a dropout layer of 50%, connecting to an output layer of size 15 with a SoftMax activation function. The 128 neuron dense layer was added because we expect the network to categorize type-related Pokemon together before going to the output layer. The dropout layer was added to prevent overfitting where 50% is a good rule of thumb to use. For the output layer, we used the SoftMax activation function because it is commonly used for classification problems where the output is a probability distribution ("tf.keras.activations.softmax", n.d.).

Using our base model, the data augmentation transformations that proved effective were horizontal reflection, translation up to 10%, zooming up to 10%, and rotation up to 10°. We then

considered sequences of pairs of those transforms, also paying attention to their order (Ex. is it better to translate then rotate or vice versa), and based on those results tested a handful of sequences with 3 or 4 transforms. The best sequence was {reflect, translate, zoom}, so we used that for all future models. See the Results section for accuracies of some of the better sequences. We then used the following model, which is inspired by Kumar (2018) who got a performance of 90% on the CIFAR-10 dataset, as well as the data augmentation described above.

```
Model: "model1_res32_no_dense_layer"
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
batch_normalization (BatchNo (None, 32, 32, 32)        128
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248
batch_normalization_1 (Batch (None, 32, 32, 32)        128
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
dropout (Dropout)            (None, 16, 16, 32)        0
conv2d_2 (Conv2D)            (None, 16, 16, 64)        18496
batch_normalization_2 (Batch (None, 16, 16, 64)        256
conv2d_3 (Conv2D)            (None, 16, 16, 64)        36928
batch_normalization_3 (Batch (None, 16, 16, 64)        256
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
dropout_1 (Dropout)          (None, 8, 8, 64)          0
conv2d_4 (Conv2D)            (None, 8, 8, 128)         73856
batch_normalization_4 (Batch (None, 8, 8, 128)         512
conv2d_5 (Conv2D)            (None, 8, 8, 128)         147584
batch_normalization_5 (Batch (None, 8, 8, 128)         512
dropout_2 (Dropout)          (None, 8, 8, 128)         0
flatten (Flatten)            (None, 8192)              0
dense (Dense)                (None, 15)                122895
```

This model also includes using a learning rate scheduler, which reduces the learning rate the longer we train since we wanted the network to converge to a local minimum stably. We also removed the final maxpooling layer to prevent excessive reduction of the final feature map's dimensions, based on prior experiments where it helped. The reasoning for this new architecture is similar to the base model reasoning. The main difference is adding another Conv2D layer before each Conv2D layer, where they all have the same settings, except that the Conv2D layers have filter sizes of 32, 32, 64, 64, 128, and 128, respectively. We also add dropout in the convolutional layers to prevent overfitting since there are more parameters being added.

Also, we initially set batch size to 256 because we wanted faster training time and expected the accuracy graphs to be smoother and less variable than using a smaller batch size

like 32. Additionally, we tested the effects of image resolution (32x32 vs 64x64) and adding a dense layer with 64 neurons and ReLU activation before the output layer. Using 32x32 images without the dense layer was most effective. We also tested the effects of changing the batch size, since Masters and Luschi (2018) and Brownlee (2020) suggest smaller sizes (ex. 32) are better. However, we found that as we decreased the batch size from 256 to 32, the accuracy steadily decreased. The accuracy also decreased for a batch size of 512, indicating that 256 is optimal. We also realized that perhaps a final maxpooling layer would be useful if the initial image size was 64x64, so we made another model with those specifications and a batch size of 256. However, this also performed worse than the 32x32 model without final maxpooling. Thus, our optimal model (see above `model1_res32_no_dense_layer`) was similar to the model in Kumar (2018), except that it has no final maxpooling layer and was trained on batch sizes of 256 (rather than 64).

Results

Data augmentation

Transformation Sequence	Test Accuracy
Translate, Zoom	0.7424390316009521
Translate, Rotate	0.730731725692749
Reflect, Translate	0.730731725692749
Reflect, Rotate	0.7229268550872803
Zoom, Translate	0.7141463160514832
Reflect, Translate, Zoom	**0.7658536434173584**
Reflect, Translate, Rotate	0.7648780345916748
Reflect, Translate, Zoom, Rotate	0.7512195110321045
Reflect, Translate, Rotate, Zoom	0.7082926630973816

The table above shows test accuracy of applying several transformation sequences to the `base_model` (sequences with low accuracy omitted), with the optimal sequence in bold. Notice how the order of transforms matters, such as {Translate, Zoom} being significantly better than {Zoom, Translate}.

Image Size and Dense Layer

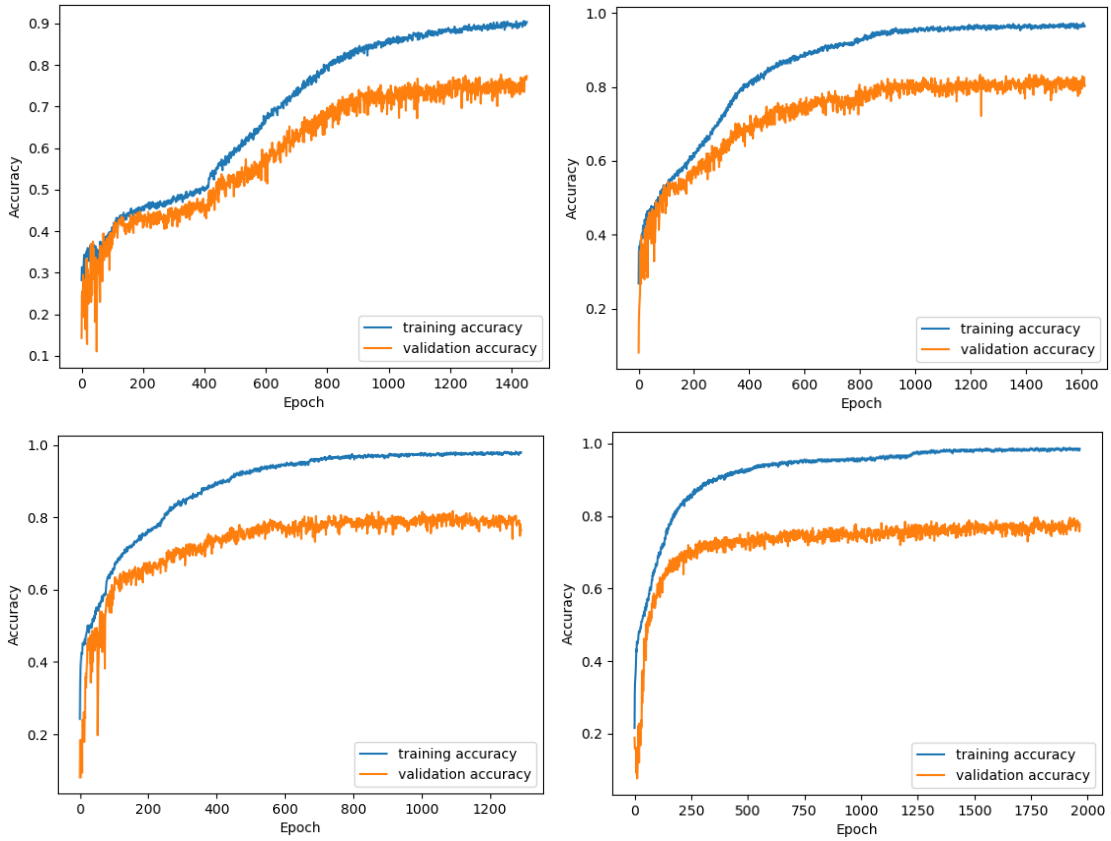
Image size \ Presence of dense layer	No	Yes
32 x 32	**0.8302438855171204**	0.6926829218864441
64 x 64	0.7248780727386475	0.37658536434173584

The table above shows test accuracy of a model like the one from Kumar (2018), with some adjustments. Refer to the Methodology section for details. The variables are the image size (32x32 vs 64x64) and the presence or absence of a final dense layer with 64 neurons. The optimal combination is bolded.

Batch size

The optimal model above used a batch size of 256. We then varied the batch size. The results for batch sizes of 64, 128, 256, and 512 are shown below, with the optimal size bolded.

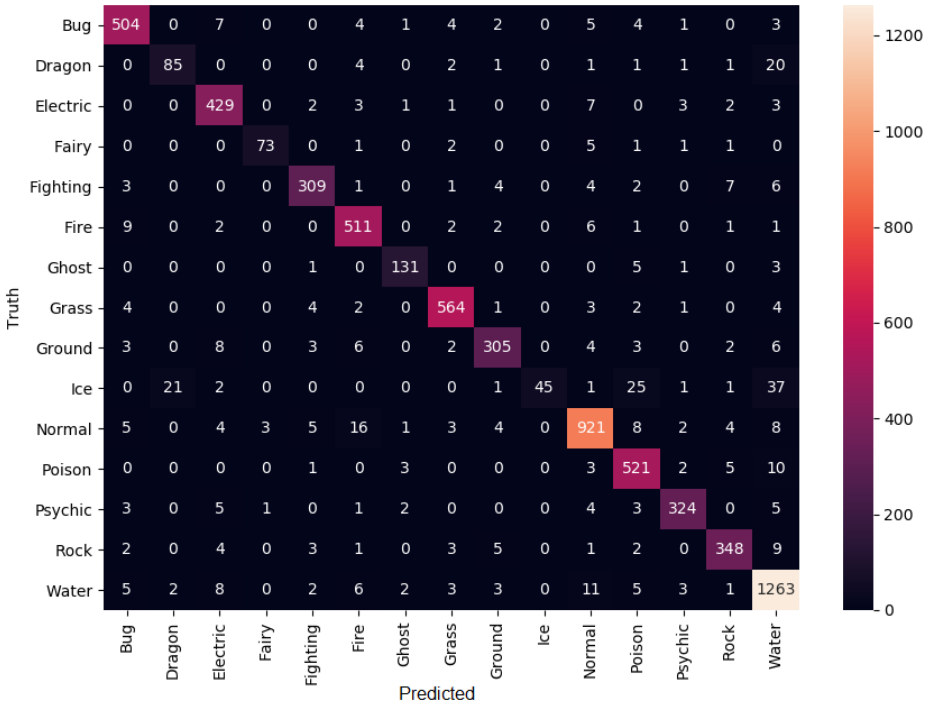
Batch Size	64	128	**256**	512
Test Accuracy	0.75414633750915	0.811707317829	**0.830243885517**	0.801951229572



Graphs of accuracy vs # of epochs for batch sizes of 64, 128, 256, and 512 respectively.

Confusion Matrix

The confusion matrix below shows which types are typically misclassified as others. It appears types with fewer images are more likely to be misclassified. For example, there are only 134 Ice types, and 37 of them (27.6%) are classified by the model as Water.



Discussion and Conclusion

Thus, the best CNN model that we managed to get that classifies the 15 unique primary types of Pokemon out of the 150 different Pokemon from the first generation through image classification is the one based on Kumar (2018) but with the final max pooling layer removed and trained on batch size of 256 instead of 64. This CNN model achieved a performance of up to 83%.

One surprise during our experimentation and results was the effect that image size had on the performance of the CNN model. For example, with the presence of a dense layer of 64 neurons in the hidden layer, training our CNN model on images with resolution sized 32x32 resulted in a performance of roughly 69%, whereas training it on images with resolution sized

64x64 resulted in a performance of approximately 38%. This is counterintuitive because we expected the larger the image resolution, the more data for the CNN to pick out important features, and hence, have higher performance. Additionally, pooling layers, which reduce the number of parameters in the convolutional layers, are not being used by state of the art CNNs such as Google Inception or ResNet, which perform very well in image classification tasks compared to traditional CNNs (Varma & Das, n.d.). Since 32x32 is a smaller resolution compared to 64x64, it is somewhat like applying pooling to our input layer. From this perspective, it feels logical to expect better performance on higher resolution images. A possible reasoning for our anomaly is that we did not train the model long enough for reasonable performance on 64x64 images or have a large enough dataset. Another surprise was the effect of batch size on accuracy. Sources like Masters and Luschi (2018) and Brownlee (2020) suggest smaller batch sizes, such as 32, are generally better. However, in our case, a batch size of 256 was optimum. This suggests that a one size fits all approach is inappropriate, and the optimal batch size may depend on other factors or parameters.

Some suggestions for further work for this image classification problem include getting more data to train on, as there are only a total of 6,829 images of 150 Pokemon in our dataset for 15 classes (15 types in generation 1), compared to CIFAR-10 which has a total of 60,000 images for its dataset classifying 10 classes. This can be done by including all generations of Pokemon which will turn the classification problem into classifying 18 classes (18 unique types) out of several hundreds of Pokemon ("Generation", 2021). In addition, to improve the performance, ResNets, or Residual Neural Networks, should be investigated and used since they are currently the state of the art in the CNN domain (Varma & Das, n.d.).

References

Subbiah, V. (2019). *Pokemon Image Dataset*. Kaggle.com. Retrieved 25 April 2021, from https://www.kaggle.com/vishalsubbiah/pokemon-images-and-types.

Zhang, L. (2020). *7,000 Labeled Pokemon*. Kaggle.com. Retrieved 25 April 2021, from https://www.kaggle.com/lantian773030/pokemonclassification.

Kumar, A. (2018). *Achieving 90% accuracy in Object Recognition Task on CIFAR-10 Dataset with Keras: Convolutional Neural Networks*. Machine Learning in Action. Retrieved 25 April 2021, from https://appliedmachinelearning.blog/2018/03/24/achieving-90-accuracy-in-object-recognition-task-on-cifar-10-dataset-with-keras-convolutional-neural-networks/.

Masters, D., & Luschi, C. (2018). *REVISITING SMALL BATCH TRAINING FOR DEEP NEURAL NETWORKS*. Arxiv.org. Retrieved 25 April 2021, from https://arxiv.org/pdf/1804.07612.pdf.

Brownlee, J. (2020). *How to Control the Stability of Training Neural Networks With the Batch Size*. Machine Learning Mastery. Retrieved 25 April 2021, from https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/.

Types. Pokémon Wiki. Retrieved 25 April 2021, from https://pokemon.fandom.com/wiki/Types.

Generation. Bulbapedia.bulbagarden.net. (2021). Retrieved 26 April 2021, from https://bulbapedia.bulbagarden.net/wiki/Generation.

Generation I. Bulbapedia.bulbagarden.net. (2021). Retrieved 25 April 2021, from https://bulbapedia.bulbagarden.net/wiki/Generation_I.

Inline Image Modification. ImageMagick. Retrieved 25 April 2021, from https://imagemagick.org/script/mogrify.php.

tf.image.resize. TensorFlow. (2021). Retrieved 25 April 2021, from https://www.tensorflow.org/api_docs/python/tf/image/resize.

tf.keras.activations.softmax. TensorFlow. Retrieved 25 April 2021, from https://www.tensorflow.org/api_docs/python/tf/keras/activations/softmax.

Stottner, T. (2019). *Why Data should be Normalized before Training a Neural Network*. towards data science. Retrieved 25 April 2021, from https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d#:~:text=Among%20the%20best%20practices%20for,and%20leads%20to%20faster%20convergence.

Varma, S., & Das, S. Chapter 12 Convolutional Neural Networks. srdas.github.io. Retrieved 25 April 2021, from https://srdas.github.io/DLBook/ConvNets.html.