CSCE 479/879 Homework 1: Classification Tasks on F-MNIST and CIFAR100

Mason Lien, Matthew Penne, Mrinal Rawool, Yves Shamavu

February 27, 2021

Abstract

Classification tasks in machine learning involve the use of algorithms to learn how to assign a class label to examples from the problem domain. Deep learning for image classification has been one of the widely studied research problems that led to the development of an ecosystem of technical artifacts like network architectures, pre-trained models and benchmark datasets. In this task, we have created models to classify two datasets F-MNIST and CIFAR100.

For the first task, we use Fashion-MNIST (F-MNIST), a well-known clothing dataset that is intended to serve as a direct drop-in replacement for the original MNIST digit dataset for benchmarking classification tasks. This assignment presented a means to explore various hyperparameters' differing adjustments to develop a robust model that generalizes well on the F-MNIST validation set. The first method of grid search exhausted 189,000 possible combinations, where cross-validation argument cycled three times through the training and validation sets to identify the combination that yielded the greatest validation accuracy. A second method of randomized search iterated to evaluate the eight different sets of hyperparameters and report the combination with the highest accuracy. The third method built upon the best model reported by the hyperparameter search approach and used weight transfer and was trained with a custom training loop. Our findings showed that the refined grid search model generalized the best over the validation set with a reported 88-90.1% accuracy with a 95% confidence interval of +/- 0.75%.

For the second task, we attempt to classify images from CIFAR100, a labeled dataset containing 60K colored images belonging to 100 different classes using two deep learning models; one based on ResNet architecture and the other based on the concept of transfer learning. Both these architectures have their advantages over a model using just convolutional layers. Our first architecture, a standalone ResNet architecture with Bottleneck layer (henceforth referred to as ResNet) achieves good performance despite having significantly fewer parameters to its convolutional counterpart with the same depth. Our second architecture, a ResNet model with transfer learning uses Xception, a pre-trained model trained on the ImageNet dataset. As Xception is also trained for an image classification task, we hope that the latter delivers better performance overall. This

exercise aims to implement and perform a comparative analysis of the performance of two model architectures. Based on our observations, the ResNet model achieved a validation accuracy of ~10%, while the transfer learning model achieved a validation accuracy of ~54%, indicating that the transfer learning model exhibits potential for generalization.

keywords: classification, Fashion-MNIST, hyperparameters, validation set, grid search, randomized search, CIFAR100, ResNet, Transfer Learning

Classification on F-MNIST dataset

1 Introduction

The Fashion-MNIST (F-MNIST) data set is well-known and routinely used for benchmarking machine learning algorithms. F-MNIST contains Zalando's article images consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes [4]. This assignment tasked us to explore an array of model tuning methods to identify which produces the highest accuracy in regards to the validation set. Grid search, randomized search, and a refined grid search with weight transfer and custom training loop were selected as the three methods of study to identify which combination of hyperparameters yielded the greatest validation accuracy.

To better understand the behavior in how these methods operate, one must know how hyperparameters and model parameters work, which directly affect model performance. Machine learning models have internal characteristics called parameters, which are values estimated from the data scanned by the model. A parameter example would be a model's beta coefficient of linear regression, or the weights associated to an artificial neural network. Model parameters are required by the model when making predictions. In contrast, a hyperparameter is an external characteristic to the model where values cannot be estimated from the data. In other words, there is no way to calculate an appropriate value. They are manually set before the model training begins. Hyperparameters are often used in helping to estimate model parameters, where they are tuned for a specific problem. The important concept to understand about hyperparameters is that the user does not know the best value to use, so trial and error and the transfer of weights from similar classification problems are the best approaches to identifying the best combination. Examples of hyperparameters used in this assignment for grid search and randomized search can be seen in Table 1.

2 Problem Description

The F-MNIST dataset is a multi-class classification problem, where the target dataset has 10 class labels (T-shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot). Multi-class classification makes the assumption that each sample is assigned to one and only one class label. This type of problem uses features extracted from the images in the dataset, where each image is then assigned one of the 10 class labels.

Table 1: Hyperparameters for Grid and Randomized Search Approaches.

	Hyperparameter	Value Range	Item Type
A	Hidden layers	6	Int
B	Neurons per layer	7	Tuple
C	Learning rate	5	Float
D	Dropout	2	Boolean
E	Dropout rate	10	Float
F	Activation	3	String / Keras object
G	Initializer	5	String / Keras object
H	Bach normalization	3	String?

3 Approaches

The approaches for the three deep learning models we have implemented are detailed in this section.

3.1 Grid search

To establish a baseline model to improve upon, we conducted two types of hyperparameter search: grid search and randomized search. Grid search is a combinatorial algorithm that evaluates all possible combinations of specified hyperparameters for each of their specified value.

For instance, in our code, we told grid search to search for hyperparameters **ABCDEFGH**, represented in Table 1, with values in the respective ranges of 6, 7, 5, 2, 10, 3, 5, 3. In this case, grid search looks through each combination of **ABCDEFGH** over every value in their respective range of values to determine which combination yields a greater accuracy.

Subsequently, grid search performed cross-validation over all the possible combinations to filter out the one with the best accuracy on the validation set. Therefore, grid search looked through 6 x 7 x 5 x 2 x 10 x 3 x 5 x 3 combinations of **ABCDEFGH**. It had explored 189,000 possible architectures of the model with different hyperparameter values. Besides, given that we gave the cross-validation argument the value of three, it meant that grid search went through 189,000 x 3 training and validation cycles.

3.2 Randomized Search

To avoid running out of computing time on the clusters before grid search exhausts its combinatorial search, we also conducted a randomized search. Randomized search is not entirely combinatorial. Instead, it tries out a select number of combinations by picking a random value from within each hyperparameter's range of values.

For instance, a randomized search would pick a value for **ABCDEFGH** within their respective value ranges (see Table 1) on a first run. It would train based on those randomly selected values. It would again pick other values for **ABCDEFGH**, still at random, and again train and evaluate their performance on a subsequent iteration.

Since we have set the number of iterations to eight, the randomized search randomly evaluated eight different values for each hyperparameter (for those whose range of values exceeded or was equal to eight - otherwise, it would pick a value more than once) and report the model that performed better on the validation set. Therefore, with a randomized search, one can control how many iterations of hyperparameter search will run based on the available computing resources.

3.3 Refined Grid Search

The grid search explores a wide combination of available hyperparameters for the model. This method proved to produce respectable accuracy, but using such course steps can overlook parameters that can yield better results. Once the large grid search provided a model with the best hyperparameters, a refined search explored hyperparameters close to those of the previous best model.

4 Experimental Setup

4.1 Preparing the dataset

The F-MNIST dataset was prepared in the following manner:

1. The Fashion-MNIST dataset is downloaded from the TensorFlow datasets
2. The training set is split into a 90 percent training and 10 percent validation.
3. The training set is shuffled 1000 and batch size for both training and validation are set at 32.
4. Both images and their labels are converted to tensors and the labels are encoded using one-hot encoding through the preprocess function.

4.2 Model Setup

1. The user is allowed to choose the type of model to train. To choose a Grid search model, set the model_to_run parameter to 1. Otherwise, a simple ResNet model is built by default.

2. The optimizer used for this project is Adam.

3. Since the learning problem is multi-class classification and the labels are one-hot encoded, the loss is measured using categorical cross-entropy.

4. The performance metric and loss metric are categorical accuracy and mean loss, respectively.

5. The file main_FMNIST.py contains the code for this set up.

6. The hyperparameters selected to explore for this assignment are shown in the following Tables [2]-[6]:

Table 2: Hidden Layers and Neurons per Layer

Hidden Layers	Number of Neurons
1	300
1,2	300, 150
1,2,3	300, 250, 150
1,2,3,4	300, 250, 150, 50
1,2,3,4	300, 300, 200, 100
1,2,3,4,5	300, 250, 150, 100, 50
1,2,3,4,5,6	300, 290, 260, 230, 200, 170

Table 3: Learning Rate

Learning Option	Rate
1	1e-5
2	1e-4
3	1e-3
4	1e-2
5	1e-1

Table 4: Dropout Rate

Dropout Option	Rate
1	0.1
2	0.2
3	0.3
4	0.4
5	0.5
6	0.6
7	0.7
8	0.8
9	0.9
10	1.0

Table 5: Activation Type

Activation Type
Rectified Linear Unit (ReLU)
Scaled Exponential Linear Unit (SELU)
Exponential Linear Unit (ELU)

Table 6: Initializer Type

Keras Initializer
he_normal
he_uniform
lecun_normal
lecun_uniform

Table 7: Grid Search Architecture

```
Model: "sequential_202"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_202 (Flatten)        (None, 784)               0
_________________________________________________________________
dense_608 (Dense)            (None, 300)               235500
_________________________________________________________________
dense_609 (Dense)            (None, 300)               90300
_________________________________________________________________
dense_610 (Dense)            (None, 200)               60200
_________________________________________________________________
dense_611 (Dense)            (None, 100)               20100
_________________________________________________________________
dense_612 (Dense)            (None, 10)                1010
=================================================================
Total params: 407,110
Trainable params: 407,110
Non-trainable params: 0
_________________________________________________________________
```

Table 8: Randomized Search Architecture

```
Model: "sequential_24"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_24 (Flatten)         (None, 784)               0
_________________________________________________________________
batch_normalization_54 (Batc (None, 784)               3136
_________________________________________________________________
dropout_42 (Dropout)         (None, 784)               0
_________________________________________________________________
dense_90 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_55 (Batc (None, 300)               1200
_________________________________________________________________
dropout_43 (Dropout)         (None, 300)               0
_________________________________________________________________
dense_91 (Dense)             (None, 250)               75250
_________________________________________________________________
batch_normalization_56 (Batc (None, 250)               1000
_________________________________________________________________
dropout_44 (Dropout)         (None, 250)               0
_________________________________________________________________
dense_92 (Dense)             (None, 150)               37650
_________________________________________________________________
batch_normalization_57 (Batc (None, 150)               600
_________________________________________________________________
dropout_45 (Dropout)         (None, 150)               0
_________________________________________________________________
dense_93 (Dense)             (None, 100)               15100
_________________________________________________________________
batch_normalization_58 (Batc (None, 100)               400
_________________________________________________________________
dropout_46 (Dropout)         (None, 100)               0
_________________________________________________________________
dense_94 (Dense)             (None, 10)                1010
=================================================================
Total params: 370,846
Trainable params: 367,678
Non-trainable params: 3,168
_________________________________________________________________
```

Table 9: Refined Grid Search with Custom Training Loop

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_198 (Flatten)        (None, 784)               0
_________________________________________________________________
dense_588 (Dense)            (None, 300)               235500
_________________________________________________________________
dense_589 (Dense)            (None, 250)               75250
_________________________________________________________________
dense_590 (Dense)            (None, 150)               37650
_________________________________________________________________
dense_591 (Dense)            (None, 50)                7550
_________________________________________________________________
batch_normalization (BatchNo (None, 50)                200
_________________________________________________________________
dense (Dense)                (None, 300)               15300
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200
_________________________________________________________________
dense_1 (Dense)              (None, 250)               75250
_________________________________________________________________
batch_normalization_2 (Batch (None, 250)               1000
_________________________________________________________________
dense_2 (Dense)              (None, 100)               25100
_________________________________________________________________
batch_normalization_3 (Batch (None, 100)               400
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1010
=================================================================
Total params: 485,510
Trainable params: 484,110
Non-trainable params: 1,400
```

5 Experimental Results

Examples of model architectures from grid search, randomized search, and refined grid search are shown in Tables 7, 8, and 9, respectively. The experimental results shown in Table 10 summarize the three different methods for model training and validation. Overall, the refined model based on grid search proved to provide the highest accuracy.

Table 10: FMNIST Architectures Ranked Based on Performance

Architecture	Validation Set Accuracy	Average Confidence Interval (95%)
Refined model based on grid search	88-90.1%	+/- 0.75%
Grid search model	88-89%	+/- 0.8%
Randomized search	86-87.5%	+/- 0.85%

Instead of reporting all three validation loss and confusion matrices for each model, below represents the proposed model with the highest accuracy, which was the refined grid search.

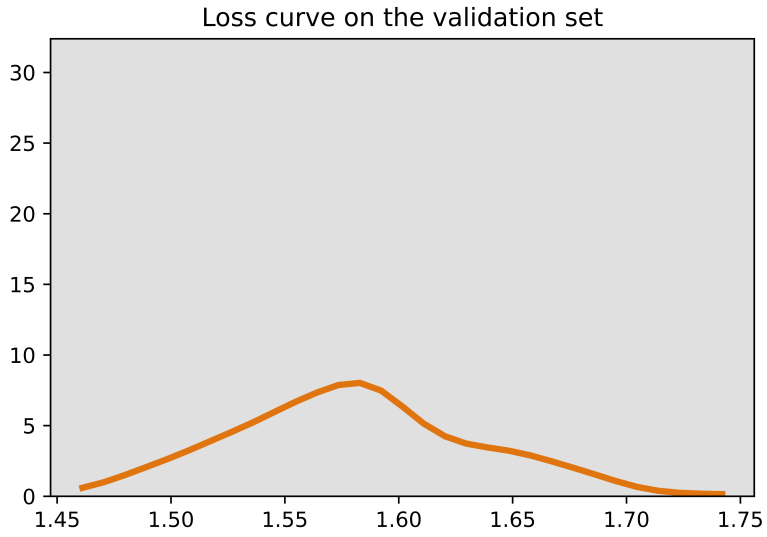

Figure 1: FMNIST Validation Loss

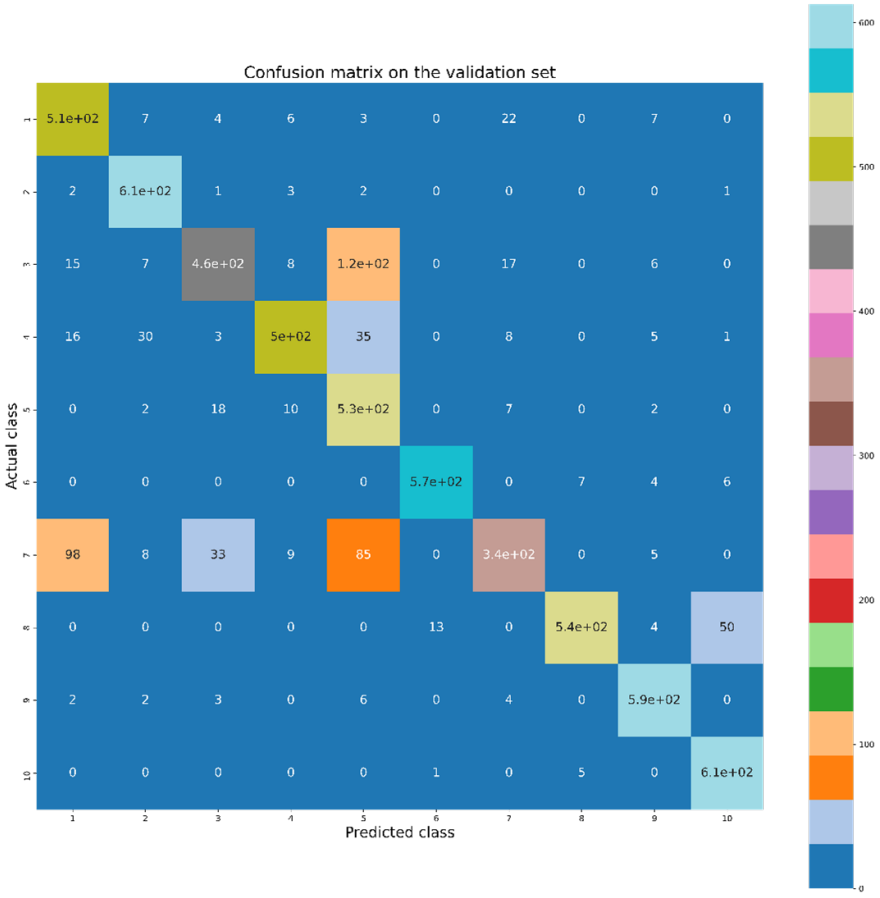

Figure 2: FMNIST Confusion Matrix

6 Discussion

This assignment tasked us to explore different methods of fine-tuning hyperparameters, which directly affected the computational cost and validation accuracy. When looking at the overall accuracy between the three methods, The refined grid search outperformed the initial grid search and the randomized search. This was due to in part to the model found through grid search had performed one to two percent better than the one from randomized search. Hence, we built upon the former and added eight additional layers below its initial architecture, thus reusing the learned weights during the grid search. This had allowed us to capture more refined patterns at the deeper levels of the architecture, and it had boosted the accuracy roughly one to one and a half percent.

Grid search is a computationally expensive method that configures optimal parameters for a given model. This method defines a search space as a grid of hyperparameter values and evaluates every position in the grid. The use of grid search is suggested for when there are combinations that are known to perform well. On the other hand, randomized search defines a search space as a bounded domain of hyperparameter values and randomly samples points in that domain. Discovery work benefits from the use of random search in how the hyperparameter combinations are defined, since pre-selected combinations from the user may not always yield the best results.

Although the refined grid search produced a model with the highest accuracy, it was essentially a two-step method that took the original information learned from grid search and applied it to the refined grid search model. When comparing computational cost, grid search required more than six hours of training time, randomized search took around two hours, and refined grid search was around seven hours in total. It is noted that if the user has limitations of computation time, then randomized search is a preferred method due to its ability to work within the set bounds to minimize the risk of running out of time or memory.

To better understand the assumptions of model accuracy, the Gaussian distribution of the proportion (classification accuracy) can be used to calculate confidence intervals (CI). The radius of the interval is calculated using the commonly used numbers of standard deviations from the Gaussian distribution. For a 95% confidence interval, the z value is 1.96. Considering the accuracy produced from refined grid search, grid search, and randomized search, the average 95% CI was +/- 0.75%, 0.8%, and 0.85%, respectively. Statistically speaking, 95 out 100 times the accuracy of a given model would fall into the reported confidence interval. For example the refined grid search reported an average validation accuracy of 89.05%, but the true classification accuracy lies between 88.3% and 89.8%. Comparing the CI for all three models, The model with the greatest interval range was randomized search, which also had the lowest reporting of validation accuracy.

Another method to summarize the performance of a classification model is through the use of a confusion matrix. This visual aid helps the user understand what the model is classifying correctly and what types of errors it is making.

Figure 2 shows the predicted vs. true labels, where the diagonal represents the same class for both predicted and true labels. Any values outside of the diagonal represent the number of miss-classifications. It is noted that class 0 (T-shirt) on the predicted axis was miss-classified as class 6 (Shirt) 98 times, followed by class 6 (Shirt) wrongly classified as class 4 (Coat) 85 times. This may be due to the reason these classes are more similar than different compared to the other classes, which make them more difficult to distinguish.

7 Conclusions

This assignment tasked us to test multiple models of grid search, randomized search, and a refined model based on the results from grid search to identify which one produced the most robust model to generalize on the validation set of the F-MNIST dataset. Our findings showed that the refined grid search model generalized the best over the validation set with a reported 88-90.1% accuracy. Since we were limited to only using dense layers in the model architectures, future work could investigate to include convolutional layers which learn spatial correlations that is a characteristic of information encoded in a given image. This would allow the model to have more information to learn from and in theory improve generalization.

Classification on CIFAR100 dataset

A Introduction

CIFAR100 is a dataset containing images belonging to 100 classes and 20 super classes. Each class contains 600 images each; 500 for training and the remaining 100 for testing. Each image is assigned two labels; one for the image class and the other one for the image super class. The training data set contains a total of 50000 images out of which 10% are set aside for validation. For every epoch, a model is trained on the 45000 training images and their corresponding labels and evaluated on the validation set. We trained these models on HCC servers and it took roughly six hours for a model to train. The model was saved using the SaveModel format to facilitate reproducibility and portability.

B Problem Description

CIFAR100 data set used for an image classification task. The model takes an image as the input, and extracts features that allow it to classify the input into one of the 100 classes the data set contains. The training dataset was loaded from Tensorflow_datasets as a Tensorflow Dataset object. The training dataset was further split into train (90%) and validation dataset (10%). Image classification

can be applied to many important tasks such as medical imaging (e.g. identification of cancer cells), agriculture (e.g. identifying crop health), education, security etc. Although, image segmentation and classification has been studied using computer vision algorithms, using deep learning models have proved to be a breakthrough that approaches the classification problem from a different perspective. Classifying images from CIFAR100 dataset is non trivial and challenging in itself owing to the presence of 100 classes. Furthermore, training a model on a large number of images can be time-consuming and computationally expensive. To alleviate these challenges, we have used two model architectures that are known to deliver excellent performance while being computationally efficient. These architectures are described in the next section.

C Approaches

The approaches for the two different architectures we designed are detailed below.

C.1 ResNet Architecture

Processing images is computationally intensive. Since images are represented as a 4D tensor, they require the use of Conv2D layers leading to creation of high dimensional feature maps. This, combined with deep network architectures result into a models with high number of parameters. This leads to challenges such as longer training times and vanishing gradients. A residual layer employs skip connections where an activation from a shallow layer is fed directly into one of the deeper hidden layers. A ResNet architecture comprising of regular Conv2D layers along with residual layers allow for all trainable layers to start learning earlier in the training phase instead of having to wait for the gradients from the shallow layers to flow through the network.

Furthermore, a ResNet architecture also uses a bottleneck layer to reduce the number of model parameters without affecting the model performance by adding a layer with a one-dimensional kernel size. This allows the network to detect more patterns along the depth axis and reduces computation. Hence, adding 1 x 1 layers around residual blocks renders the latter as thin as possible, thereby increasing the network's depth without expanding the number of parameters. Furthermore, the 1X1 kernels also allows the input and output feature maps dimensions to be adjusted for compatibility with the preceding and succeeding layers. The architecture is shown in Fig 3.

C.2 Transfer Learning

By default, the weights and biases of a neural network are initialized randomly. Transfer learning is a useful technique to improve a model's performance while potentially reducing the training time. This is done by using a pre-trained model as the base layer for a new model. In order to fully benefit from this technique,

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, None, 32, 32, 3)]	0
zero_padding2d (ZeroPadding2	multiple	0
conv2d (Conv2D)	multiple	9408
batch_normalization (BatchNo	multiple	256
activation (Activation)	multiple	0
max_pooling2d (MaxPooling2D)	multiple	0
residual_unit (ResidualUnit)	multiple	74240
global_average_pooling2d (Gl	multiple	0
flatten (Flatten)	multiple	0
dense (Dense)	multiple	65000
dropout (Dropout)	multiple	0
dense_1 (Dense)	multiple	100100

Total params: 249,004
Trainable params: 248,620
Non-trainable params: 384

Figure 3: Model: ResNet with Bottleneck layers

the pre-trained model has to be trained on a similar dataset or a similar task or both.

For this task, it was challenging to train a model anew given the number of classes and the features' quality due to the low-resolution images. Although our base model's architecture implements all the bells and whistles of a state-of-the-art deep learning model, including dropout layers, residual layers, etc., the model took a prolonged time to train. To counter this challenge, we created a second model architecture that used pre-trained layers from the variant of the GoogLeNet called Xception. It was proposed by Chollet in 2016 [1], and it includes both residual and bottleneck blocks, as well as a new type of block called depthwise separable convolution. This block analyses the depth and spatial components separately wise and the spatial Reduces the computational complexity without affecting the model performance. For example, to get a 256 channel output feature map for a 32X32 RGB image using 2D convolution kernels of shape 3X3, the total number of computations = 30 X 30 X 27 X 256 = 6,220,800. However, when using a depth wise separable convolution block,

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
custom_preprocess (CustomPre (None, 224, 224, 3)       0
_________________________________________________________________
xception (Functional)        (None, None, None, 2048)  20861480
_________________________________________________________________
global_average_pooling2d (Gl (None, 2048)              0
_________________________________________________________________
dropout (Dropout)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 100)               204900
=================================================================
Total params: 21,066,380
Trainable params: 204,900
Non-trainable params: 20,861,480
```

Figure 4: Model: ResNet with Transfer Learning

each of the RGB channels in an image undergo convolution with filter channels separately (hence depth wise separated) to get three 30 X 30 X 1 feature maps that are stacked together to form 30 X 30 X 3 feature map. Up to this point, the image was analyzed spatially. When this 30 X 30 X 3 image undergoes a depth wise convolution with a 1 X 1 X 3 kernel, it produces a 30 X 30 X 1 feature map. Therefore, if the 30 X 30 X 3 feature map undergoes convolutions with 256 1 X 1 X 3 kernels, it results into a 30 X 30 X 256 feature map. However, the number of computations required to achieve this is 715,500 which is significantly less than the computations required for creating the same feature map using a convolutional layer. The architecture for the model with transfer learning is shown in Fig 4.

D Experimental Setup

The set-up
1. Train and Validation sets are loaded using the Tensorflow Dataset API.
2. The train and Validation datasets are further split into batches of size 32. Both images and their labels are converted to tensors. The labels are encoded using one-hot encoding.
3. Next, the user is allowed to choose the type of model to train. To choose a ResNet model with transfer learning, set the model_to_run parameter to 2. Otherwise, a simple ResNet model is built by default.
4. The optimizer used for this project is Adam.
5. Since the learning problem is multi-class classification and the labels are

one-hot encoded, the loss is measured using categorical cross-entropy.
6. The performance and loss metrics are categorical accuracy and mean loss, respectively.
7. The model is trained by a custom training loop.
8. The file main_CIFAR.py contains the code for this set-up.

E Experimental Results

Summary of Model Training

Table 11: Model Training Summary

Description	ResNet	Transfer Learning
Total time to train:	around 5-6 hours	around 5-6 hours
Final training accuracy	99%	99%
Final validation	10%	52%
Final training loss	<0.1	<0.1
Final validation loss	18	3

F Discussion

The results of training the models is shown in Table 11. Both models displayed overfitting during the training phase. However, the overfitting was more pronounced in case of the ResNet model. The accuracy of ResNet on training data was $> 99\%$, while that on validation data was $< 10\%$. On the other hand, accuracy of the transfer learning model was also $> 99\%$, while the validation accuracy hovered around 53% to 55%. The length of the model training process is a major area of improvement for future work.

G Conclusions

One of the goal of this exercise was to observe whether the transfer learning model delivers superior performance compared to the ResNet model. Based on the validation accuracy observed during training, the transfer learning model shows the potential to generalize better than the ResNet model. The CIFAR learning task also shed light on some technical challenges our team faced during the model training process. Both the models took 6 hours to train and greatly impacted our ability to modify our experiments and understand the impact of those modifications. Our team also tried to work around these challenges by leveraging MATLAB to train the models. However, the difference in the model formats on MATLAB and Tensorflow precluded us from taking advantage of the speed and efficiency of MATLAB. Lessons from the CIFAR100 task will be a guiding factor in our future work.

A Comparison Between Matlab and Tensorflow

A.1 Introduction

During the course of this assignment severe weather caused electricity shortages, resulting in the use of HCC to be suspended. This prompted the group to attempt training neural networks on their personal machines. The large data structures and complex architectures used general have prohibitively long training times when using a laptop, or even a respectable desktop. The group has prior experience with the conventionally used python and the academically used Matlab. In several of the group members anecdotal experience, Matlab is faster than Python in both matrix math and loop structures, i.e. the basis of deep neural networks. The group attempted to train complicated CNN architectures for the CIFAR-100 dataset in Matlab and export to Python, but this proved difficult beyond the scope of the class. To salvage some work and to test a hypothesis, the group trained the same CNN architecture to the CIFAR-100 dataset in both Python and Matlab and are compared below. The personal machine used for the model training used an i7-7700 at 3.6 GHz, a Nvidia 1070, and 16 GB of ram.

A.2 Python Architecture and Training

The CIFAR-10 dataset is a commonly used dataset for teaching deep CNN, and the CIFAR-100 is just a more complicated version. The rough outline for the CNN is shown below in Fig. 5 with help from [3]. The architecture used sequential convolutions layers with increasing filters of 32, 64, 128. The layers used relu activation, with batch normalization followed by a max pooling layer. The final layer was a flattened, then connected to a dense layer with 100 neurons and softmax activation. The training used Adam optimization, categorical cross entropy loss functions, and trained for 30 epochs. The input data was randomly horizontally rotated, split in to 80% training data and 20% validation data, and used batch sizes of 128 images. The training produced a validation accuracy of 35.95% and a loss of 3.678 whose training curves are shown in Fig. 6, and overall the training took 18.24 minutes.

A.3 Matlab implementation

Again, the same CNN architecture, data preparation, and training was repeated in Matlab. The Matlab provided reference of [2] proved useful. The validation accuracy was 47.84%, the training loss was 2.9191, with the training curves shown in 7. The overall training time was 9.63 minutes.

A.4 Discussion

The goal of this side project was to prove that Matlab is faster than tensorflow for training a given network architecture. Not only is Matlab twice as fast as

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 30, 30, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 13, 13, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 128)         73856
_________________________________________________________________
batch_normalization_2 (Batch (None, 4, 4, 128)         512
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 1, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 100)               12900
=================================================================
Total params: 107,044
Trainable params: 106,596
Non-trainable params: 448
```

Figure 5: Matlab validation and error curves

tensorflow, it also produced significantly high accuracy. This may be attributed to how Matlab handles the Adam optimization, weight initialization, batch normalization, and weight updates, compared to python. Both models several over fit the data, which could be expected from such a complex data set.

Tensorflow has the advantage of being free and widely supported. Many project are done in tensorflow that could be readily adapted to various applications. But Matlab is significantly faster and has potential to produce better models in a given architecture. If one has access to Matlab and is training a model from scratch Matlab is worth considering.

A.5 Exporting Matlab Models

The starting goal of this side project was to train a model in Matlab and export it to Python, but that proved difficult. The main way to export a Matlab CNN

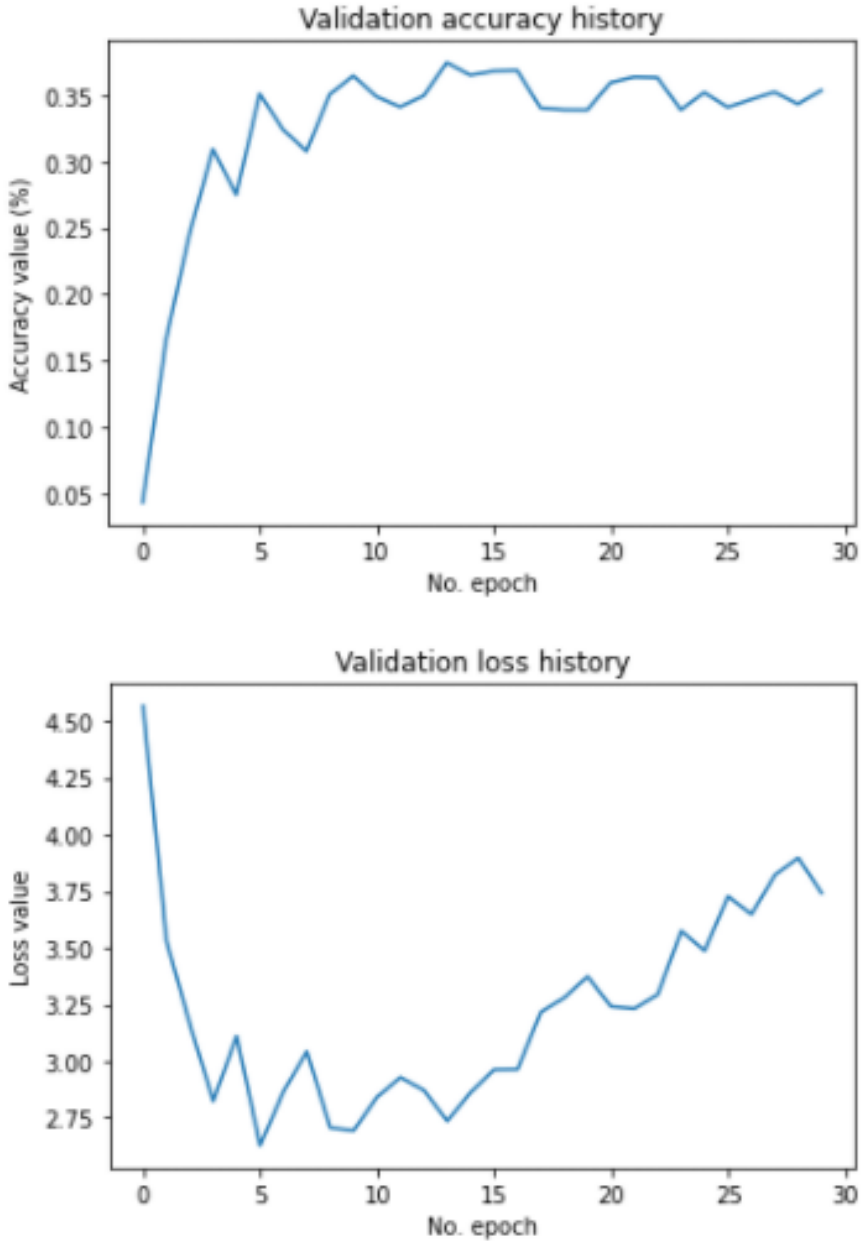

Figure 6: Matlab validation and error curves

model is with the *exportONNXnetwork()* function. This exports the model to the onnx deep learning standard. The problem is that onnx uses the NCHW

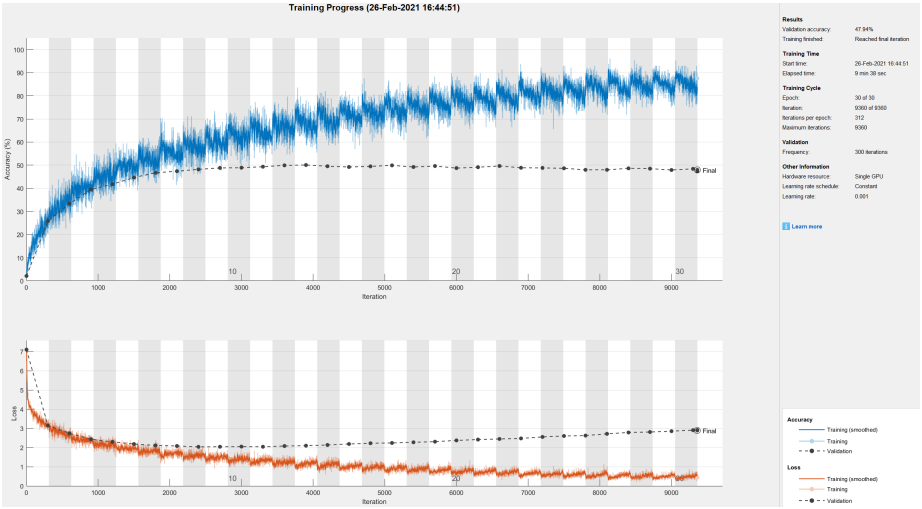

Figure 7: Matlab validation and error curves

(number-channels-height-width) format instead of the tensorflow NHWC format. Converting from NCHW to NHWC appears to be its own area of research and is not widely available. Another option is to read in the Matlab CNN weights individually, but this again proved difficult. In the future, this process will hopefully become easier so one can take advantage of the speed of Matlab and the support of Tensorflow.

References

[1] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[2] Inc. Mathworks. Train residual network for image classification, 2020. URL: `www.mathworks.com/help/deeplearning/ug/train-residual-network-for-image-classification.html`.

[3] Christian Versloot. How to build a convnet for cifar-10 and cifar-100 classification with keras, 2020. URL: `www.machinecurve.com/index.php/2020/02/09/how-to-build-a-convnet-for-cifar-10-and-cifar-100-classification-with-keras/`.

[4] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.