Machine Learning- COIY065H7 - Coursework Submission

Ryan Hill (13151863)
rhill06@mail.bkk.ac.uk
r.l.hill128@gmail.com
Wordcount: 3203

April 9, 2020

Declaration

Contents

1 Introduction

1.1 Weight–wise Adaptive learning rates with Moving average Estimator

The weight-wise adaptive learning rates with moving average estimator (WAME) algorithm, first proposed by Mosca et al.,[6] is an algorithm that can be used in place of the current choice of deep learning optimizers including Adam and RMSProp. The algorithm combines the approaches of Rprop and RMSProp, using both the sign of the product of the current and previous gradient, as well as an exponentially weighted moving average (EWMA) factor, θ, to tackle the vanishing gradient problem. Their contribution is to take the idea of a dynamic learning rate and extend this to have a weight-wise adaptive learning rate i.e. an updating learning rate for every weight within the network. The impact of this, reported by the authors, is that the training loss of a network decreases at a faster rate, and, within the first 100 epochs of their data, to a lower value than the current popular optimizers. The benefit of this approach over traditional approaches is that weights of the network that quickly tend to a local minima are able to be refined sooner, whilst those that are further from a minima are able to continue to take large changes, both without impacting on the other.

Details of the algorithm and the parameters are available within their paper, but we present here the adjusted algorithm used in this work after discussion with the author.

Algorithm 1 Adjusted WAME Algorithm

```
1:  procedure WAME($\alpha, \eta_+, \eta_-, \zeta_{min}, \zeta_{max}, \lambda$)        ▷ Hyperparameters to be chosen by user
2:      $\theta_{ij}(0) = 0, Z_{ij}(0) = 0, \zeta_{ij}(0) = 1 | \forall i, j$
3:      for all $t \in [1..T]$ do
4:          if $\frac{\partial E(t)}{\partial w_{ij}} \times \frac{\partial E(t-1)}{\partial w_{ij}} > 0$ then
5:              $\zeta_{ij}(t) = \min\{\zeta_{ij}(t-1) \times \eta_+, \zeta_{max}\}$
6:          else if $\frac{\partial E(t)}{\partial w_{ij}} \times \frac{\partial E(t-1)}{\partial w_{ij}} < 0$ then
7:              $\zeta_{ij}(t) = \max\{\zeta_{ij}(t-1) \times \eta_-, \zeta_{min}\}$
8:          else
9:              $\zeta_{ij}(t) = \zeta_{ij}(t-1)$
10:         end if
11:         $Z_{ij}(t) = \alpha Z_{ij}(t-1) + (1-\alpha)\zeta_{ij}(t)$        ▷ EWMA of the acceleration factor
12:         $\theta_{ij}(t) = \alpha\theta_{ij}(t-1) + (1-\alpha)\left(\frac{\partial E(t)}{\partial w_{ij}}\right)^2$        ▷ EWMA of the RMSProp divisor
13:         $\Delta w_{ij}(t) = -\lambda Z_{ij}(t) \frac{\partial E(t)}{\partial w_{ij}} \frac{1}{\sqrt{\theta_{ij}(t) + \epsilon}}$
14:         $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$
15:     end for
16: end procedure
```

where λ is the learning rate, $\{\zeta_{min}, \zeta_{max}\}$ are clipping values for the acceleration factor ζ, α is the exponential decay rate, $\{\eta_+, \eta_-\}$ are additional hyperparameters, and ϵ is some system precision small value to avoid division by zero. The changes made from the original algorithm are the explicit inclusion of ϵ and the square root of the θ in line 13 to reduce runaway weight changes, plus the clarification of an

else case.

We implemented the algorithm in Tensorflow's OptimizerV2 class[9] as this is a purer implementation than the using the keras API meaning it was more likely to work with some of the more experimental features of Tensorflow/Keras such as the use of TPUs (which ended up not being required) and a certainty of working with kerastuner as mentioned in section [2.1]. The implementation is detailed in listing [1] and follows the standard class strucutre for a V2 optimiser. The initialisation of the class instantiates all the hyperparameters for the instance of the class specifically as hyperparameters for Tensorflow, and the epsilon fudge factor. *_create_slots* is used to create additional tensors within the class that can be referenced and updated throughout the training, and then *_prepare_local* creates a set of constant tensors that we will reference in the alogrithm such as the clipping values and both alpha and 1-alpha. The bulk of the algorithm is implemented in *_resource_apply_dense* which is equivalent to one pass of the for loop within the algorithm, updating the variables as it goes before finally returning the updated weights of the network. This method differs most from the Keras optimizer implementation which instead uses an explicit for loop, and the pure Tensorflow approach uses wrapper functions to C++ operations which can be seen within the code. We chose not to implement the sparse version of this function as we would not explicitly be dealing with sparse tensors naturally in this work. The final component of the class is the *get_config* method which is required to pass information about the class back to the user and is trivial.

```
class WAMEprop(optimizer_v2.OptimizerV2):
    """WAME optimizer.
    It is recommended to leave the parameters of this optimizer at their default
   values as these have been
    shown empircally to deliver good results (except the learning rate, which can be
     freely tuned).
    The algorithm has been adapated slightly from the original paper to replace \
   frac{1}{\theta} with \frac{1}{\sqrt{\theta}} after
    speaking with the algorithm developers.
    # Arguments
        learning_rate: float >= 0. Base learning rate.
        alpha: float >= 0. Decay rate of the exponentially weighted moving average.
        eta_plus: float > 0. Multiplicative term of the acceleration factor for the
   case of a positive gradient product.
        eta_minus: float > 0. Multiplicative term of the acceleration factor for the
   case of a negative gradient product.
        zeta_min: float > 0. Lower bounding value for the accerlation factor.
        zeta_max: float > 0. Upper bounding value for the acceleration factor.
        epsilon: float > 0. A very small fudge factor requried to avoid a possible
   division by zero error.
    # References
        - [wame: Training Convolutional Networks with -Weightwise Adaptive Learning
   Rates]
        (https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-50.pdf)
    """

    def __init__(self, learning_rate=0.0001, alpha = 0.9, eta_plus = 1.2, eta_minus
   = 0.1, zeta_min = 0.01, zeta_max = 100, epsilon = 1e-11, **kwargs):
        super(WAMEprop, self).__init__(**kwargs)
        self._set_hyper("learning_rate", kwargs.get("lr", learning_rate))
        self._set_hyper("alpha", alpha)
        self._set_hyper("eta_plus", eta_plus)
        self._set_hyper("eta_minus", eta_minus)
        self._set_hyper("zeta_min", zeta_min)
        self._set_hyper("zeta_max", zeta_max)
        self.epsilon = epsilon

    def _create_slots(self, var_list):
        for var in var_list:
          self.add_slot(var, "zetas")
          self.add_slot(var, "zeds")
          self.add_slot(var, "thetas")
          self.add_slot(var, "old_grads")

    def _prepare_local(self, var_device, var_dtype, apply_state):
        super(WAMEprop, self)._prepare_local(var_device, var_dtype, apply_state)

        alpha = array_ops.identity(self._get_hyper("alpha", var_dtype))
        eta_plus = array_ops.identity(self._get_hyper("eta_plus", var_dtype))
        eta_minus = array_ops.identity(self._get_hyper("eta_minus", var_dtype))
        zeta_max = array_ops.identity(self._get_hyper("zeta_max", var_dtype))
        zeta_min = array_ops.identity(self._get_hyper("zeta_min", var_dtype))
        apply_state[(var_device, var_dtype)].update(
          dict(
            epsilon=ops.convert_to_tensor_v2(self.epsilon, var_dtype),
            alpha=alpha,
            eta_plus = eta_plus,
            eta_minus = eta_minus,
            zeta_max = zeta_max,
            zeta_min = zeta_min,
            one_minus_alpha = 1 - alpha))

    def _resource_apply_dense(self, grad, var, apply_state=None):
        var_device, var_dtype = var.device, var.dtype.base_dtype
        coefficients = ((apply_state or {}).get((var_device, var_dtype))
                or self._fallback_apply_state(var_device, var_dtype))

        zeta =  self.get_slot(var, 'zetas')
        zed =  self.get_slot(var, 'zeds')
        theta =  self.get_slot(var, 'thetas')
        old_grad =  self.get_slot(var, 'old_grads')

        new_z = tf.where(
          math_ops.Equal(x = grad * old_grad, y = 0),
        zeta,
        tf.where(math_ops.Greater(x = grad * old_grad, y = 0),
          x = math_ops.Minimum(x = zeta * coefficients['eta_plus'], y = coefficients['
   zeta_max']),
          y = math_ops.Maximum(x = zeta * coefficients['eta_minus'], y = coefficients[
      'zeta_min'])
        )
        )

        new_z = state_ops.assign(zeta, new_z, use_locking=self._use_locking)

        new_zed = (coefficients["alpha"] * zed)  + (coefficients["one_minus_alpha"]*
   new_z)
        new_zed = state_ops.assign(zed, new_zed, use_locking=self._use_locking)

        new_t = (coefficients["alpha"] * theta)  + (coefficients["one_minus_alpha"]*
   math_ops.square(grad))
        new_t = state_ops.assign(theta, new_t, use_locking=self._use_locking)

        var_t  = var - (coefficients["lr_t"] * new_zed * grad * (1/(math_ops.Sqrt(x =
   new_t) +coefficients["epsilon"])))

        old_grad = state_ops.assign(old_grad, grad, use_locking=self._use_locking)

        return state_ops.assign(var, var_t, use_locking=self._use_locking).op

    def _resource_apply_sparse(self, grad, var):
        raise NotImplementedError("Sparse␣gradient␣updates␣are␣not␣supported.")

    def get_config(self):
        config = super(WAMEprop, self).get_config()
        config.update({'learning_rate': self._serialize_hyperparameter("learning_rate"
   ),
           'alpha': self._serialize_hyperparameter("alpha"),
           'eta_plus': self._serialize_hyperparameter("eta_plus"),
           'eta_minus': self._serialize_hyperparameter("eta_minus"),
           'zeta_min': self._serialize_hyperparameter("zeta_min"),
           'zeta_max': self._serialize_hyperparameter("zeta_max")
           })
        return config
```

Listing 1: WAME class implementation

1.2 EMNIST Letters

The EMNIST (Extended Modified National Institute of Standards and Technology) set is a collection of datasets created to build on the success of the standard MNSIT digits dataset for image recognition benchmarking and address the need for a more complex, yet still easy to understand and debug, dataset for more powerful models that achieve $\geq 99\%$ accuracy on MNSIT. Details of the processing and sets of data available are detailed by Cohen et al[2] in their paper, but in short the processing applied to these images was as close as possible to the original processing completed for the MNIST dataset. The EMNIST set contains multiple subsets of the data, we have chosen to use the *letters* dataset which consists of 26 classes with no

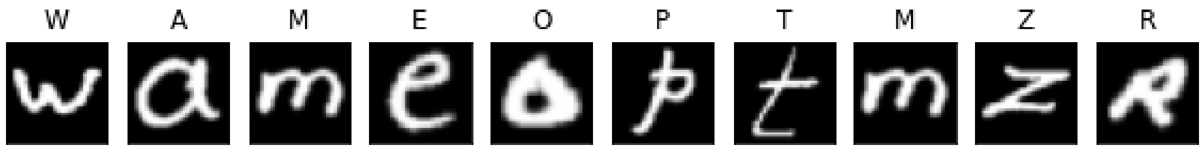


Figure 1.1: Example of 10 images within the EMNIST letters dataset

distinction between upper and lowercase letters, meaning not only is it a more complex problem in terms of the number of target classes compared to the 10 in MNIST, but also has the additional challenge of recognising 2 often different characters and mapping these to the same output. An example of the images within the dataset (after restoring them to the correct reflection and orientation) is shown in fig. [1.1].

This data comes by default in a training and test set, with 124800 and 20800 balanced records of the 26 classes respectively. However, the training set also has a balanced validation set appended to the end of it with 20800 records so we remove this from the training set explicitly to use for validation later. It is not clear from the source paper what the mix of upper and lowercase letters within this dataset is, it is possible the mix if 50/50 but it is not obvious from the wording.

1.3 Google Colab

The work detailed herein has been completed on the Google Colab[4] system which is a currently free-to-use jupyter-like instance to allow people to run python code without the constraints of their machines. In particular, they make available GPU and TPU instances for order of magnitude speed up to model training. However, due to the fact this is meant to be used by the likes of students and independent researchers, there is no way to identify exactly what hardware a piece of code was ran on, or even that the same GPU is used throughout an instance, so we cannot report hardware specifics for this work.

2 Methodology and Design

2.1 Usage of Keras Tuner

We have chosen to use the relatively new *kerastuner* package due to the advantages it provides over the manual approach of human trial and error. By removing as much of the human element from the tuning as possible it allows us to ensure that we are not injecting too much bias in what we believe will be the correct choice.

The package offers out of the box the choice of 3 tuning algorithms, RandomSearch, Baysian search, and the HyperBand algorithm. RandomSearch is the most basic of those offered, with the only value it provides over manual trial and error is that it will automatically look within the provided search space up to some maximum number of trails without having to specify every combination you wish to check, but rather just the limits and step sizes. Baysian search and HyperBand both offer an improvement over RandomSearch and we chose to go with HyperBand for the reasons explained within the next section.

That package allows for the user to specify, as mentioned, upper and lower limits for pretty much any part of the model the user wishes, including but not limited to the learning rate or other training parameters, the number of dense layers, or the number of neurons within those layers. Once these values are provided the algorithms will automatically identify the provided search space and, depending on what method was chosen, attempt to optimise its search time to find the best resulting model.

2.1.1 The HyperBand Algorithm

The HyperBand algorithm is, at its heart, a time optimised random search algorithm. The *Band* stands for bandit, who's goal it is to maximise their *profit* in a given time. First proposed by Li et al. in 2018[5] this algorithm was proposed as an alternative to Baysian methods as it had been shown that running a RandomSearch for twice as long was likely to produce better results than the Baysian approach, thus if this RandomSearch could be optimised in some way then better results could be found in the same or even less time.

The paper itself is over 30 pages long, with a further nearly 20 pages of appendixes of detailed experimental results and various proofs and theorems but the basic idea behind the algorithm itself is quite simple; rather than running every tested combination of hyperparameters through to completion as RandomSearch does, only run the model for a few epochs before culling a proportion of the population. This is then repeated with a period of more epochs for the now smaller population of possible models, and another cull is then made; over and over until just the best result remains. Less computing time is wasted on models that don't appear, at the start at least, to produce impactful results. This matches the human approach, if we were manually tuning these models it is possible that we would stop a training run if we saw low performance and time was an issue. Given infinite computing resource time we would have no need for this, but unfortunately we don't have that so this is a sensible approach. For this work we keep the default of a reduction in population size to one third at each step, and a 3 fold increase in the number of epochs at each step.

There is one key drawback of this method; it will favour models which improve quickly, and will *throw away* population members who might eventually outperform other models, but would take more epochs to reach the same level initially. This is even more important to understand when learning rate is one of the parameters being tuned; in fact one could argue that, given many top performing models on sites such as Kaggle are often XGBoost models with low learning rates and high iterations, learning rate should not be tuned via this approach at all. The authors argue that this is negated by hedging, however in the default configuration of the algorithm very few of the population actually run through to completion so with limited resources this still remains an issue. For this work we still choose to tune on the learning rate as our goal was not to make a direct comparison between different models, but to try with limited resources to find the best model we could.

2.2 Model design and Hyperparameters for tuning

As the EMNIST dataset is the extension of the MNIST data, we chose to keep the same design for the feature extraction part of the Convolutional Neural Network (CNN) as presented in the paper by Mosca et al.,[6] that is 2 convolutional layers of size 64 with strides of 5x5 and 1x1 respectively, followed by a 2x2 max pooling layer, followed by the same setup except the convolutional layers have size 128; all with $relu$ activation. We chose to keep this design as it has shown already to be more than capable at extracting

Hyper Parameter	**Min Value**	**Max Value**	**Step/sampling**
# Dense Layers	1	5	1
Neurons per dense layer	256	1024	256
Dropout rate post dense layer	0.4	0.6	0.1
Learning rate	0.001	0.1	log

Table 2.1: Configuration of tune-able parameters for the CNN

the features required in such a simple and low resolution dataset, even though the number of classes has increased the features used to identify and classify numbers should be similar enough to letters.

The classification part of the network is where we chose to spend the majority of our tuning, along with the learning rate for the optimizer. The hyperparameters we chose to vary are given in table [2.1] but are summarised as using between 1 and 5 densely connected layers with between 256 and 1024 neurons per layer, followed by dropout of between 0.4 and 0.6 for normalisation to hopefully increase generalisability. The learning rate was the only hyperparameter of the optimizer itself we chose to vary, between 1×10^{-1} and 1×10^{-3} using log sampling to take into account the usual approach for tuning learning rate, leaving all other values as suggested in the original paper. Overall this leads to quite a large search space which supports the choice of using a pre-built tuner.

We note at this time that due to the time taken to train and test each model, we have chosen to use the simple validation set approach to identify the best configuration of the model as opposed to e.g. k-fold cross validation. We set the tuner to run each trial twice to attempt to mitigate the impact of random starts, but we cannot rule out the possibility that the tuner results will not be entirely consistent outside of this seed. To mitigate this we will manually create and train the 5 best models and look at both their validation accuracy as well as their test accuracy to choose our *best* configuration.

3 Experimental Results

3.1 Tuner Results and extended testing

The tuner in total ran 90 different trials, with 2 executions per trial to help reduce the impact of random starts. This number could have been made higher but given the limited computational time we had this was deemed a reasonable approximation. The results of the top 5 configurations from the tuner are available in table [3.1] and we created these models to be able to extract more information from them. We trained the model using the same batch size with a maximum of 500 epochs, with an early stopping applied if we saw no further increase in the accuracy on the validation dataset after 25 epochs (i.e. a patience of 25), and saved the best model from within this period with regards to the same measure. The Top-1 training, validation, and test accuracy across all classes, as well as the epoch the model was taken from is also available in the aforementioned table. The training accuracy in this should not be taken as the best possible for that configuration as this was the accuracy at the time of the best model for validation accuracy i.e. those with more epochs are likely to have a better training accuracy regardless. The per-class results are available in our results file but are not discussed here for brevity. Overall it is clear that configuration 2, which was a

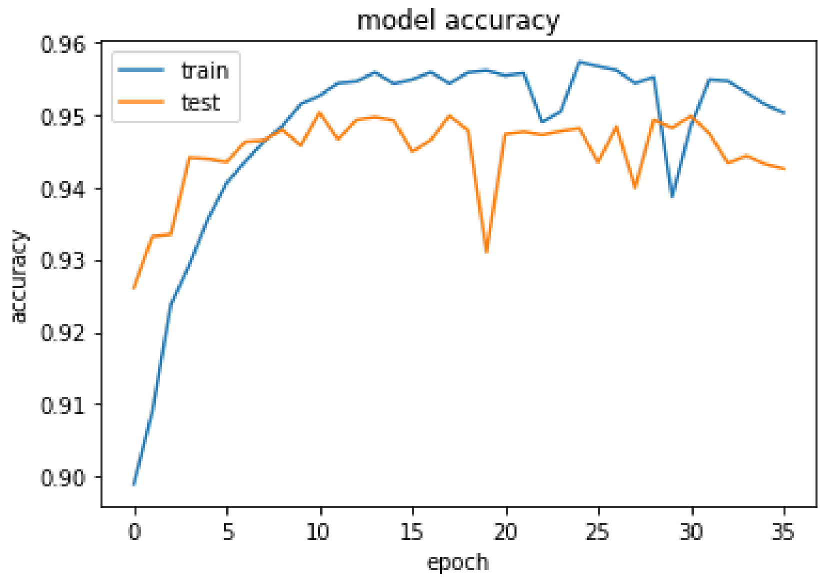


Figure 3.1: Training history of configuration 1. Here test refers to the validation dataset

close second in the tuner, has performed best in all accuracy categories, whilst taking more epochs to train than configuration 1 these are still small numbers and the cost of training these networks will not be an order of magnitude different. The results of this model are analysed in the next section. An example of the training progression of one of these configurations can be seen in fig. [3.1].

3.2 Final model results

The final model (best based on validation and test accuracy from the previous section), when evaluated on the unseen test set resulted in an overall accuracy of 94.6%, increasing to 99.1% for Top-2 classification and to 99.5% for Top-3. The confusion matrix for the test data and Top-1 predicted class is presented in fig. [3.2] and per-class recall results for Top 1, 3, and 5 predictions are detailed within table [3.2]. Further insight into these results is available within the next section.

3.3 Discussion

Overall the classifier performed well, improving on the benchmark of 85% given by Cohen's OPIUM classifier,[2] however we do see the same types of mistakes as they did. Baldominos et al[1] have conveniently completed a survey of results on this dataset in 2019 and we can use their work to compare how our model performs. Of the 12 (6 classical, 6 involving CNNs) classifiers reported in their survey for the letters dataset, we see that only 3 of these models out-performed our results, and even then by no more than 1% overall. These classifiers were all somewhat more advanced in their design than ours, using either Markov random field concepts, neuroevolution, or capsules. Given the additional complexity of these models it is fair to say that our classifier performed strongly.

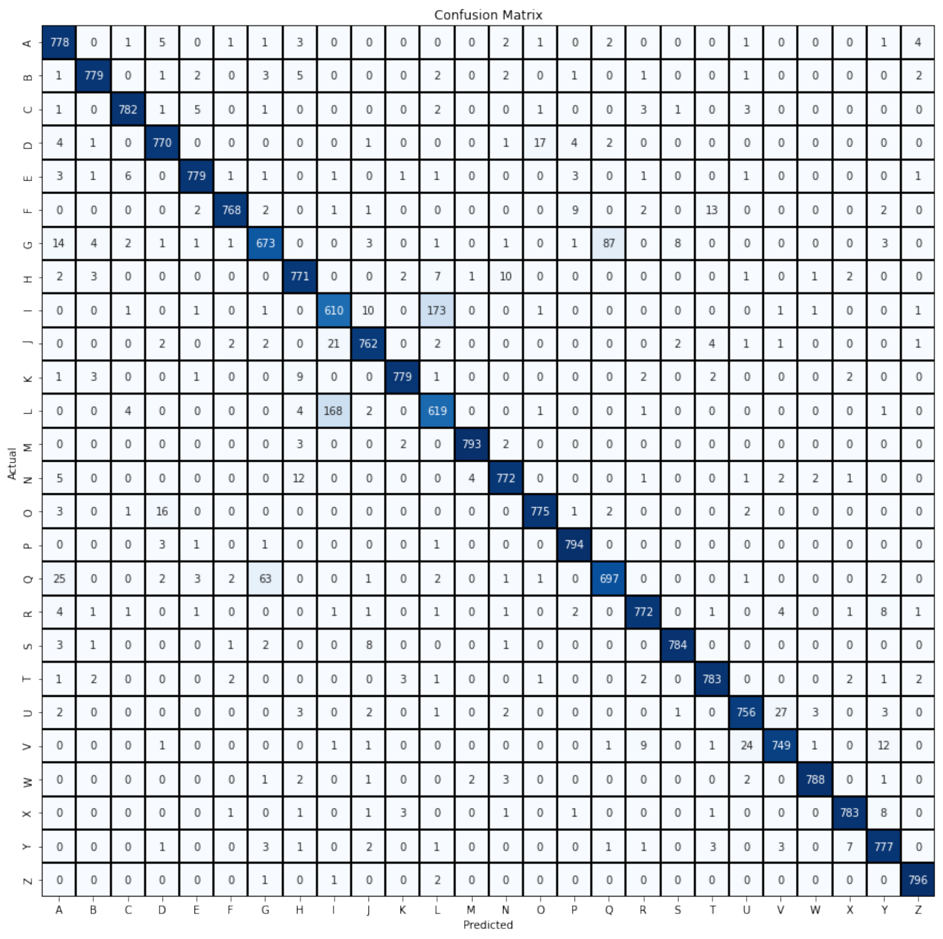
The large jump in accuracy between top-1 and top-2/3 results indicate a specific failure in classification that can be easily identified when looking at the confusion matrix. The largest non-diagonal elements are

Model Rank*	**1**	**2**	**3**	**4**	**5**
Number of layers	2	2	3	4	2
# Neurons	(768, 768)	(512, 512)	(512, 768, 768)	(768, 512, 256, 512)	(256, 256)
Dropout	(0.6, 0.6)	(0.4, 0.4)	(0.4, 0.5, 0.6)	(0.5, 0.5, 0.6, 0.5)	(0.4, 0.5)
Learning Rate	0.0755	0.0524	0.0217	0.0229	0.0725
Tuner Result (%)	**94.9567**	94.9182	94.8990	94.8966	94.8870
Training Accuracy (%)	96.6548	**97.3971**	97.3711	96.8538	96.8548
Validation Accuracy (%)	95.0288	**95.2067**	94.9951	94.8701	94.8173
Best Epoch^	11	30	30	26	17
Test Accuracy (%)	94.6346	**94.6586**	94.5961	94.5240	94.5769

Table 3.1: Results of the best 5 model configurations from the tuner and manual production of the models. Best results for each accuracy category are in bold.

*Rank based on the validation accuracy reported by the tuner

^Best epoch based on the epoch with the best validation accuracy with an early stopping with a patience of 25



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	778	0	1	5	0	1	1	3	0	0	0	0	0	2	1	0	2	0	0	0	1	0	0	0	1	4
B	1	779	0	1	2	0	3	5	0	0	0	2	0	2	0	1	0	1	0	0	1	0	0	0	0	2
C	1	0	782	1	5	0	1	0	0	0	0	2	0	0	1	0	0	3	1	0	3	0	0	0	0	0
D	4	1	0	770	0	0	0	0	0	1	0	0	0	1	17	4	2	0	0	0	0	0	0	0	0	0
E	3	1	6	0	779	1	1	0	1	0	1	1	0	0	0	3	0	1	0	0	1	0	0	0	0	1
F	0	0	0	0	2	768	2	0	1	1	0	0	0	0	0	9	0	2	0	13	0	0	0	0	2	0
G	14	4	2	1	1	1	673	0	0	3	0	1	0	1	0	1	87	0	8	0	0	0	0	0	3	0
H	2	3	0	0	0	0	0	771	0	0	2	7	1	10	0	0	0	0	0	0	1	0	1	2	0	0
I	0	0	1	0	1	0	1	0	610	10	0	173	0	0	1	0	0	0	0	0	0	1	1	0	0	1
J	0	0	0	2	0	2	2	0	21	762	0	2	0	0	0	0	0	0	2	4	1	1	0	0	0	1
K	1	3	0	0	1	0	0	9	0	0	779	1	0	0	0	0	0	2	0	2	0	0	0	2	0	0
L	0	0	4	0	0	0	0	4	168	2	0	619	0	0	1	0	0	1	0	0	0	0	0	0	1	0
M	0	0	0	0	0	0	0	3	0	0	2	0	793	2	0	0	0	0	0	0	0	0	0	0	0	0
N	5	0	0	0	0	0	0	12	0	0	0	0	4	772	0	0	0	1	0	0	1	2	2	1	0	0
O	3	0	1	16	0	0	0	0	0	0	0	0	0	0	775	1	2	0	0	0	2	0	0	0	0	0
P	0	0	0	3	1	0	1	0	0	0	0	1	0	0	0	794	0	0	0	0	0	0	0	0	0	0
Q	25	0	0	2	3	2	63	0	0	1	0	2	0	1	1	0	697	0	0	0	1	0	0	0	2	0
R	4	1	1	0	1	0	0	0	1	1	0	1	0	1	0	2	0	772	0	1	0	4	0	1	8	1
S	3	1	0	0	0	1	2	0	0	8	0	0	0	1	0	0	0	0	784	0	0	0	0	0	0	0
T	1	2	0	0	0	2	0	0	0	0	3	1	0	0	1	0	0	2	0	783	0	0	0	2	1	2
U	2	0	0	0	0	0	0	3	0	2	0	1	0	2	0	0	0	0	1	0	756	27	3	0	3	0
V	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	1	9	0	1	24	749	1	0	12	0
W	0	0	0	0	0	0	1	2	0	1	0	0	2	3	0	0	0	0	0	0	2	0	788	0	1	0
X	0	0	0	0	0	1	0	1	0	1	3	0	0	1	0	1	0	0	0	1	0	0	0	783	8	0
Y	0	0	0	1	0	0	3	1	0	2	0	1	0	0	0	0	1	1	0	3	0	3	0	7	777	0
Z	0	0	0	0	0	0	1	0	1	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	796

Figure 3.2: Confusion matrix of final model

Class	**Top 1**	**Top 3**	**Top 5**
A	0.97250	0.99625	0.99750
B	0.97375	0.99250	0.99375
C	0.97750	0.99250	0.99750
D	0.96250	0.99750	0.99875
E	0.97375	0.99000	0.99500
F	0.96000	0.99125	0.99625
G	0.84125	0.99500	0.99500
H	0.96375	0.99625	1.00000
I	0.76250	0.99250	0.99500
J	0.95250	0.99500	0.99875
K	0.97375	0.99500	0.99750
L	0.77375	0.99875	0.99875
M	0.99125	0.99625	0.99875
N	0.96500	0.99750	0.99875
O	0.96875	0.99375	0.99375
P	0.99250	0.99625	0.99875
Q	0.87125	0.98875	0.99250
R	0.96500	0.99500	0.99875
S	0.98000	0.99250	0.99625
T	0.97875	0.99500	0.99625
U	0.94500	0.99375	0.99500
V	0.93625	0.99250	0.99375
W	0.98500	0.99375	0.99500
X	0.97875	0.99750	0.99875
Y	0.97125	0.99750	1.00000
Z	0.99500	0.99625	0.99875

Table 3.2: Per class recall of the model for Top 1/3/5

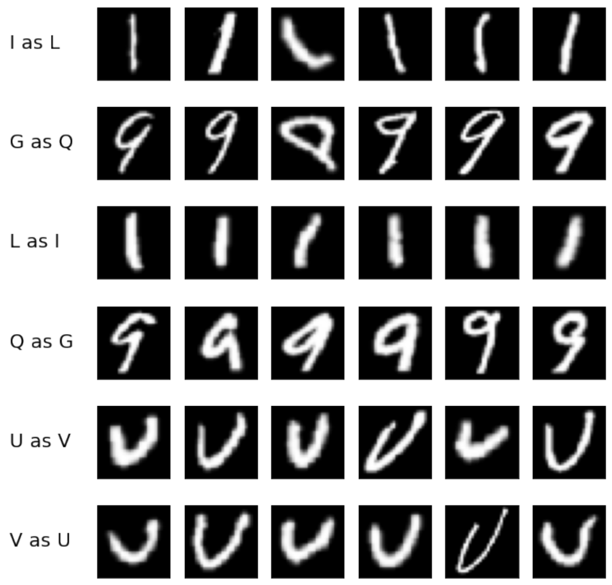


Figure 3.3: Examples of misclassified letters. Each row represents a different combination of the most common actual as predicted errors i.e. row 1 is records that are actually Is, but were misclassified as Ls

at the intersections of I and L, G and Q, and U and V. Whilst other errors do occur, these are by far the most prevalent ones. It is easy to rationalise these mistakes, especially when we recall that the letters dataset combines both upper and lowercase letters, making especially the I and L combination almost impossible for even a human to determine. Figure [3.3] displays 6 examples from each pairing in both directions where misclassification took place.

Once we look at the Top-3 performance per class, we see that only 1 class still fall below a 99% accuracy, Q, but only falls just below this. Overall this model, while not perfect in isolation for single handwritten character recognition, could be used for an optical character recognition (OCR) system designed to digitise documents, as given context of a whole word it would be possible to check online which combination of high probability letters lead to a real word in that language for cases where multiple letters have a high class probability.

4 Conclusion

In this work we implemented the WAME algorithm via the Tensorflow package and their custom optimizer set up. We then used this in the kerastuner package to test a variety of model configurations on the EMNIST letters dataset, before taking the 5 best performing configurations and comparing those directly. The best model produced via this method gave a Top-1 accuracy of 95.65% and a Top-3 accuracy of 99.5%, results up there with some of the best performing models on this dataset we could identify results for, whilst being somewhat more simplistic in the design.

There is more work that could be done on the model, in particular being able to train for longer with more patience in the early stopping implementation could improve results over the long term, although this is likely to only be marginal. There is an intrinsic difficulty as seen in this dataset with the combination of upper and lowercase letters, which means there is realistically an upper ceiling that a model could hope to achieve. An expansion of the tuner search space could also be considered, either by changing the max/min and steps of the selected hyperparameters, varying other hyperparameters of the optimizer, or even tuning the convolutional part of the network itself.

If a substantive amount of work was to be undertaken, then the use of ensemble methods could be considered to see if improvement could be gained there. Alternatively, as discussed earlier, the tuning method favours models that make improvement quickly from the start, there could be work done to explore the space of slowly improving models, although this would require additional computing resource.

Beyond these improvements, the application of this model combined with other information could be used in OCR with a high degree of accuracy. We have presented a near state-of-the-art classifier using a novel optimizer and proven its use within deep learning model training.

References

[1] A. Baldominos, Y. Saez, and P. Isasi, “A Survey of Handwritten Character Recognition with MNIST and EMNIST,” *Applied Sciences*, vol. 9, no. 15, p. 3169, aug 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/15/3169

[2] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, “EMNIST: an extension of MNIST to handwritten letters,” feb 2017. [Online]. Available: http://arxiv.org/abs/1702.05373

[3] B. Descamps, “Custom Optimizer in TensorFlow - Towards Data Science.” [Online]. Available: https://towardsdatascience.com/custom-optimizer-in-tensorflow-d5b41f75644a

[4] Google, “Colaboratory – Google.” [Online]. Available: https://research.google.com/colaboratory/faq.html

[5] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization,” *Journal of Machine Learning Research*, vol. 18, pp. 1–52, mar 2018. [Online]. Available: http://jmlr.org/papers/v18/16-558.html.http://arxiv.org/abs/1603.06560

[6] A. Mosca and G. D. Magoulas, “Training Convolutional Networks with Weight-wise Adaptive Learning Rates,” in *ESANN 2017*, 2017. [Online]. Available: http://www.i6doc.com/en/.

[7] J. Prost, “How to Perform Hyperparameter Tuning with Keras Tuner.” [Online]. Available: https://www.sicara.ai/blog/hyperparameter-tuning-keras-tuner

[8] Srijan14, “srijan14 Implementation of Keras Handwritten Chracter Recognition.” [Online]. Available: https://github.com/srijan14/keras-handwritten-character-recognition/blob/master/src/model.py

[9] Tensorflow, “Tensorflow optimizer_v2.” [Online]. Available: https://github.com/tensorflow/tensorflow/tree/v2.1.0/tensorflow/python/keras/optimizer{_}v2

A Package Versions

Core packages used in the work are detailed in table [A.1] below. As previously mentioned due to the use of Google Colab it is not possible to specify the exact hardware the code was run on.

Package	**Version**
Python	3.6.9
numpy	1.18.2
pandas	0.25.3
pickle	4.0
tensorflow	2.1.0
keras	2.3.0
scipy	1.4.1
matplotlib	3.2.0
kerastuner	1.0.1

Table A.1: Python and package version numbers

B Full Code

Whilst the full code is provided here, it is recommended to view this via the original code in the notebook state as submitted with this work.

```
# -*- coding: utf-8 -*-
"""ML CW.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1VY7m_5NB35i6Ss3kCLkqobKumApVxxE_

## Install packages required
"""

# Commented out IPython magic to ensure Python compatibility.
# %tensorflow_version 2.x
!pip install tensorflow==2.1.0
!pip install keras==2.3.0
!pip install -U keras-tuner

"""## Create the WAME optimizer for use within the later training
First we create the WAME optimizer within Tensorflow using the existing optimizers
   as a framework to build upon.
"""

from tensorflow.python.framework import ops
from tensorflow.python.keras import backend_config
from tensorflow.python.keras.optimizer_v2 import optimizer_v2
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import control_flow_ops
from tensorflow.python.ops import math_ops
from tensorflow.python.ops import state_ops
from tensorflow.python.training import training_ops
from tensorflow.python.util.tf_export import keras_export
import numpy as np
import tensorflow as tf
import random

import pandas as pd
from scipy.io import loadmat
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

from tensorflow.keras.layers import Dense, Activation, Dropout, Flatten,
 MaxPooling2D, BatchNormalization, Reshape
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.models import Sequential
from kerastuner.tuners import *
from kerastuner.engine.hypermodel import HyperModel
from kerastuner.engine.hyperparameters import HyperParameters
import pickle
import h5py

from sklearn.metrics import confusion_matrix, classification_report,
 accuracy_score
import string
import seaborn as sn


class WAMEprop(optimizer_v2.OptimizerV2):
  """WAME optimizer.
  It is recommended to leave the parameters of this optimizer at their default
 values as these have been
  shown empircally to deliver good results (except the learning rate, which can be
   freely tuned).
  The algorithm has been adapated slightly from the original paper to replace \
 frac{1}{\theta} with \frac{1}{\sqrt{\theta}} after
  speaking with the algorithm developers.
  # Arguments
     learning_rate: float >= 0. Base learning rate.
     alpha: float >= 0. Decay rate of the exponentially weighted moving average.
     eta_plus: float > 0. Multiplicative term of the acceleration factor for the
 case of a positive gradient product.
     eta_minus: float > 0. Multiplicative term of the acceleration factor for the
 case of a negative gradient product.
```

```
    zeta_min: float > 0. Lower bounding value for the accerlation factor.
    zeta_max: float > 0. Upper bounding value for the acceleration factor.
    epsilon: float > 0. A very small fudge factor requried to avoid a possible
division by zero error.
  # References
    - [wame: Training Convolutional Networks with -Weightwise Adaptive Learning
Rates]
    (https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2017-50.pdf)
  """

  def __init__(self, learning_rate=0.0001, alpha = 0.9, eta_plus = 1.2, eta_minus
= 0.1, zeta_min = 0.01, zeta_max = 100, epsilon = 1e-11, **kwargs):
    super(WAMEprop, self).__init__(**kwargs)
    self._set_hyper("learning_rate", kwargs.get("lr", learning_rate))
    self._set_hyper("alpha", alpha)
    self._set_hyper("eta_plus", eta_plus)
    self._set_hyper("eta_minus", eta_minus)
    self._set_hyper("zeta_min", zeta_min)
    self._set_hyper("zeta_max", zeta_max)
    self.epsilon = epsilon

  def _create_slots(self, var_list):
    for var in var_list:
      self.add_slot(var, "zetas")
      self.add_slot(var, "zeds")
      self.add_slot(var, "thetas")
      self.add_slot(var, "old_grads")

  def _prepare_local(self, var_device, var_dtype, apply_state):
    super(WAMEprop, self)._prepare_local(var_device, var_dtype, apply_state)

    alpha = array_ops.identity(self._get_hyper("alpha", var_dtype))
    eta_plus = array_ops.identity(self._get_hyper("eta_plus", var_dtype))
    eta_minus = array_ops.identity(self._get_hyper("eta_minus", var_dtype))
    zeta_max = array_ops.identity(self._get_hyper("zeta_max", var_dtype))
    zeta_min = array_ops.identity(self._get_hyper("zeta_min", var_dtype))
    apply_state[(var_device, var_dtype)].update(
      dict(
        epsilon=ops.convert_to_tensor_v2(self.epsilon, var_dtype),
        alpha=alpha,
        eta_plus = eta_plus,
        eta_minus = eta_minus,
        zeta_max = zeta_max,
        zeta_min = zeta_min,
        one_minus_alpha = 1 - alpha))

  def _resource_apply_dense(self, grad, var, apply_state=None):
    var_device, var_dtype = var.device, var.dtype.base_dtype
    coefficients = ((apply_state or {}).get((var_device, var_dtype))
            or self._fallback_apply_state(var_device, var_dtype))

    zeta =  self.get_slot(var, 'zetas')
    zed =  self.get_slot(var, 'zeds')
    theta =  self.get_slot(var, 'thetas')
    old_grad =  self.get_slot(var, 'old_grads')

    new_z = tf.where(
      math_ops.Equal(x = grad * old_grad, y = 0),
    zeta,
    tf.where(math_ops.Greater(x = grad * old_grad, y = 0),
      x = math_ops.Minimum(x = zeta * coefficients['eta_plus'], y = coefficients['
zeta_max']),
      y = math_ops.Maximum(x = zeta * coefficients['eta_minus'], y = coefficients[
'zeta_min'])
    )
    )

    new_z = state_ops.assign(zeta, new_z, use_locking=self._use_locking)

    new_zed = (coefficients["alpha"] * zed)  + (coefficients["one_minus_alpha"]*
new_z)
    new_zed = state_ops.assign(zed, new_zed, use_locking=self._use_locking)

    new_t = (coefficients["alpha"] * theta)  + (coefficients["one_minus_alpha"]*
math_ops.square(grad))
    new_t = state_ops.assign(theta, new_t, use_locking=self._use_locking)

    var_t  = var - (coefficients["lr_t"] * new_zed * grad * (1/(math_ops.Sqrt(x =
new_t) +coefficients["epsilon"])))

    old_grad = state_ops.assign(old_grad, grad, use_locking=self._use_locking)

    return state_ops.assign(var, var_t, use_locking=self._use_locking).op

  def _resource_apply_sparse(self, grad, var):
    raise NotImplementedError("Sparse gradient updates are not supported.")

  def get_config(self):
    config = super(WAMEprop, self).get_config()
    config.update({'learning_rate': self._serialize_hyperparameter("learning_rate"
),
          'alpha': self._serialize_hyperparameter("alpha"),
          'eta_plus': self._serialize_hyperparameter("eta_plus"),
          'eta_minus': self._serialize_hyperparameter("eta_minus"),
          'zeta_min': self._serialize_hyperparameter("zeta_min"),
          'zeta_max': self._serialize_hyperparameter("zeta_max")
          })
    return config
```

```
"""## Import data and split
Due to the data being stored in the a matlab file, we take the majority of this
 import code from https://github.com/srijan14/keras-handwritten-character-
 recognition/blob/master/src/model.py to simplify the approach and do the
 required pre-processing to rotate and transpose the data.
"""

def load_data(file, img_rows = 28, img_cols = 28, log = False):
  letters = loadmat(file)
  # Loading Training Data
  X_train = letters["dataset"][0][0][0][0][0][0]
  y_train = letters["dataset"][0][0][0][0][0][1]
  X_train = X_train.astype('float32')
  X_train /= 255.0

  ##Loading Testing Data
  X_test = letters["dataset"][0][0][1][0][0][0]
  y_test = letters["dataset"][0][0][1][0][0][1]
  X_test = X_test.astype('float32')
  X_test /= 255.0

  # one-hot encoding:
  Y_train = to_categorical(y_train - 1)
  Y_test = to_categorical(y_test - 1)

  # input image dimensions
  X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
  X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)

  # Reshaping all images into 28*28 for pre-processing
  # MNIST and EMNIST are all rotated and transposed for some weird reason...
  X_train = X_train.reshape(X_train.shape[0], 28, 28)
  X_test = X_test.reshape(X_test.shape[0], 28, 28)
  # for train data
  for t in range(X_train.shape[0]):
    X_train[t] = np.transpose(X_train[t])

  # for test data
  for t in range(X_test.shape[0]):
    X_test[t] = np.transpose(X_test[t])

  # Reshape the train data
  X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
  X_train = X_train.reshape(X_train.shape[0], 784, )

  # Split out the validation set
  X_valid = X_train[-20800:,:]
  Y_valid = Y_train[-20800:,:]
  X_train = X_train[:-20800,:]
  Y_train = Y_train[:-20800,:]

  # Reshape the test data
  X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
  X_test = X_test.reshape(X_test.shape[0], 784, )

  if log:
    print('EMNIST data loaded: train:', len(X_train), 'Validation:', len(X_valid),
  'test:', len(X_test))
    print('Flattened X_train:', X_train.shape)
    print('Y_train:', Y_train.shape)
    print('Flattened X_valid:', X_valid.shape)
    print('Y_valid:', Y_valid.shape)
    print('Flattened X_test:', X_test.shape)
    print('Y_test:', Y_test.shape)

  return X_train, Y_train, X_valid, Y_valid, X_test, Y_test

X_train, Y_train, X_valid, Y_valid, X_test, Y_test =  load_data('/content/drive/My
 Drive/emnist-letters.mat', log = True)

letters = [22, 0, 12, 4, 14, 15, 19, 12, 25, 17]

letter_locs = [np.where(np.argmax(Y_train, axis = 1) == i)[0][2] for i in letters]

#wameoptmzr

fig, ax = plt.subplots(1, 10, sharex='col', sharey='row', figsize=(10, 1))
for j in range(10):
  data = X_train[letter_locs[j]].reshape((28, 28))
  ax[j].imshow(data, cmap='gray')
  ax[j].tick_params(left=False, labelleft=False, bottom = False, labelbottom =
 False)
  ax[j].title.set_text('{label}'.format(label=string.ascii_uppercase[np.argmax(
 Y_train[letter_locs[j]])]))

"""## Create the model and tuner
Here we create the class to call a tuneable model, and import all the required
 functions and objects to run the tuning itself.
"""

class MyHyperCNN(HyperModel):

  def __init__(self, num_classes):
    self.num_classes = num_classes

  def build(self, hp):
    model = Sequential()
    model.add(Reshape((28, 28, 1), input_shape=(784,)))
    model.add(Conv2D(64, (5, 5), input_shape=(28, 28, 1), activation = 'relu'))
    model.add(Conv2D(64, (1, 1), activation = 'relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(128, (5, 5), activation = 'relu'))
    model.add(Conv2D(128, (1, 1), activation = 'relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())

    # Fully connected layer
    for i in range(hp.Int('num_layers', min_value = 1, max_value = 5, step = 1)):
        model.add(Dense(units=hp.Int('units_' + str(i),
                                                   min_value=256,
                                                   max_value=1024,
                                                   step=256),
                                  activation='relu'))
        model.add(Dropout(hp.Float('units_drop_out' + str(i),
                                                   min_value=0.4,
                                                   max_value=0.6,
                                                   step=0.1)))

    model.add(Dense(self.num_classes))
    model.add(Activation('softmax'))

    model.compile(optimizer= WAMEprop(learning_rate = hp.Float(
                        'learning_rate',
                        min_value=1e-3,
                        max_value=1e-1,
                        sampling='log',
                        default=1e-2
                    ), name = 'wame'),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
    return model

"""## Create and run the tuner
First we create the tuner and check the search space, and then we run the tuner
 model itself, saving the output incase we get disconnected.
"""

tuner = Hyperband(
    MyHyperCNN(26),
    max_epochs=30,
    objective='val_accuracy',
    seed=123,
    executions_per_trial=2,
    directory='hyperband',
    project_name='emnist_letters',
)

tuner.search_space_summary()

tuner.search(X_train,
          Y_train,
          batch_size = 1000,
          validation_data=(X_valid, Y_valid),
          verbose = 0
          )

tuner.results_summary()

pickle.dump(tuner, open( "/content/drive/My Drive/tuner1.p", "wb" ) )

tuner.results_summary()
print('####################################')
tuner.get_best_hyperparameters()[0].values

"""## Tuner Analysis"""

tuner =pickle.load(open("/content/drive/My Drive/tuner1.p", "rb" ))
n_trails = 0
try:
  while True:
    temp = tuner.get_best_hyperparameters(10000000)[n_trails].values
    n_trails += 1
except:
  print("There were", n_trails, "trials")



tuner.results_summary()

"""## Create, train, evaluate final model
Here we create the final model, hard coding in the best parameters from the tuner,
  train the model, and then evaluate it on the test dataset.
"""

def make_final_model(num_layers, n_neurons, dropouts, lr):
  model = Sequential()
  model.add(Reshape((28, 28, 1), input_shape=(784,)))
  model.add(Conv2D(64, (5, 5), input_shape=(28, 28, 1), activation = 'relu'))
  model.add(Conv2D(64, (1, 1), activation = 'relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  model.add(Conv2D(128, (5, 5), activation = 'relu'))
  model.add(Conv2D(128, (1, 1), activation = 'relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  model.add(Flatten())

  # Fully connected layer
  for i in range(num_layers):
    model.add(Dense(n_neurons[i]))
    model.add(Activation('relu'))
    model.add(Dropout(dropouts[i]))

  model.add(Dense(26))
  model.add(Activation('softmax'))

  model.compile(optimizer= WAMEprop(learning_rate = lr, name = 'wame'),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
  return model

models = [make_final_model(2, [768, 768], [0.6, 0.6], 0.07554544167531015),
          make_final_model(2, [512, 512], [0.4, 0.4], 0.0524485340749127),
          make_final_model(3, [512, 768, 768], [0.4, 0.5, 0.6],
 0.021788030540912 46),
          make_final_model(4, [768, 512, 256, 512], [0.5, 0.5, 0.6, 0.5],
 0.02291990786900 5517),
          make_final_model(2, [256, 256], [0.4, 0.5], 0.07255565349251887)
]

model_names = []
for i in range(5):
  model_names.append('Best_' + str(i))

histories = []
for model, name in zip(models, model_names):
  print('Running model', name)
  tf.random.set_seed(123)
  random.seed(456)
  np.random.seed(789)

  #early stopping and checkpoints
  es = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', verbose=1,
 patience=25)
  mc = tf.keras.callbacks.ModelCheckpoint('/content/drive/My Drive/
 model_experiment_' + name +'.h5', monitor='val_accuracy', mode='max', verbose=1,
  save_best_only=True, save_weights_only=True,)

  histories.append(model.fit(X_train,
            Y_train,
            epochs = 500,
            batch_size = 1000,
            validation_data=(X_valid, Y_valid),
            callbacks = [es, mc],
            verbose = 0)
  )

#visualise one of the model history
plt.plot(histories[0].history['accuracy'])
plt.plot(histories[0].history['val_accuracy'])
plt.title('model␣accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper␣left')
plt.show()

"""## All model basic eval"""

models = [make_final_model(2, [768, 768], [0.6, 0.6], 0.07554544167531015),
          make_final_model(2, [512, 512], [0.4, 0.4], 0.0524485340749127),
          make_final_model(3, [512, 768, 768], [0.4, 0.5, 0.6],
 0.02178803054091246),
          make_final_model(4, [768, 512, 256, 512], [0.5, 0.5, 0.6, 0.5],
 0.022919907869005517),
          make_final_model(2, [256, 256], [0.4, 0.5], 0.07255565349251887)
]

model_names = []
for i in range(5):
  model_names.append('Best_' + str(i))

for model, name in zip(models, model_names):
  model.load_weights('/content/drive/My␣Drive/model_experiment_' + name +'.h5')

y_preds_train = []
y_preds_val = []
y_preds_test = []
for model in models:
  y_preds_train.append(model.predict_classes(X_train))
  y_preds_val.append(model.predict_classes(X_valid))
  y_preds_test.append(model.predict_classes(X_test))

for train, val, test, i in zip(y_preds_train, y_preds_val, y_preds_test, range(5)
 ):
  print('Model', i, ':' 'Train:', accuracy_score(np.argmax(Y_train, axis = 1),
 train),
      'Val:',accuracy_score(np.argmax(Y_valid, axis = 1), val),
      'Test:',accuracy_score(np.argmax(Y_test, axis = 1), test))

"""## Model Evaluation"""
```

```
model = make_final_model(2, [512, 512], [0.4, 0.4], 0.0524485340749127)
model.load_weights('/content/drive/My Drive/model_experiment_Best_1.h5')

y_pred = model.predict_classes(X_test)
y_pred_prob = model.predict_proba(X_test)

print(accuracy_score(np.argmax(Y_test, axis = 1), y_pred))

def top_n_accuracy(preds, truths, n):
  """ Thank you stackoverflow https://stackoverflow.com/questions/32461246/how-to-
 get-top-3-or-top-n-predictions-using-sklearns-sgdclassifier/48572046"""
  best_n = np.argsort(preds, axis=1)[:,-n:]
  ts = np.argmax(truths, axis=1)
  successes = 0
  for i in range(ts.shape[0]):
    if ts[i] in best_n[i,:]:
      successes += 1
  return float(successes)/ts.shape[0]

def top_n_recall_per_class(preds, truths, n, classes):
  n_classes = len(classes)
  best_n = np.argsort(preds, axis=1)[:,-n:]
  ts = np.argmax(truths, axis=1)
  successes = [0]*n_classes
  class_count = [0]*n_classes
  for i in range(ts.shape[0]):
    class_count[ts[i]] += 1
    if ts[i] in best_n[i,:]:
      successes[ts[i]] += 1
  return {k:v for k, v in zip(classes, [float(i)/float(j) for i, j in zip(
 successes, class_count)])}

recall1 = top_n_recall_per_class(y_pred_prob, Y_test, 1, string.ascii_uppercase)
recall3 =top_n_recall_per_class(y_pred_prob, Y_test, 3, string.ascii_uppercase)
recall5 =top_n_recall_per_class(y_pred_prob, Y_test, 5, string.ascii_uppercase)
rc1_df = pd.DataFrame.from_dict(recall1, orient = 'index', columns = ['Top 1'])
rc3_df = pd.DataFrame.from_dict(recall3, orient = 'index', columns = ['Top 3'])
rc5_df = pd.DataFrame.from_dict(recall5, orient = 'index', columns = ['Top 5'])
pd.concat([rc1_df, rc3_df, rc5_df], axis = 1)

accuracy = []
for i in range(1, 6):
  accuracy.append(top_n_accuracy(y_pred_prob, Y_test, i))
print(accuracy)
plt.plot(range(1, 6), accuracy)
plt.title('Test accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Top N')
plt.show()
```

```
cm = confusion_matrix(np.argmax(Y_test, axis = 1), y_pred)

df_cm = pd.DataFrame(cm, index = [i for i in string.ascii_uppercase],
                  columns = [i for i in string.ascii_uppercase])
plt.figure(figsize = (15,15))
g = sn.heatmap(df_cm, annot=True, fmt='g', cmap = plt.cm.Blues, linewidths= 1,
  linecolor = 'black', cbar= False)
plt.title('Confusion Matrix')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()

misclassified_i_as_l = np.where((np.argmax(Y_test, axis = 1) != y_pred) & (np.
 argmax(Y_test, axis = 1) == 8) & (y_pred == 11))
misclassified_g_as_q = np.where((np.argmax(Y_test, axis = 1) != y_pred) & (np.
 argmax(Y_test, axis = 1) == 6) & (y_pred == 16))
misclassified_l_as_i = np.where((np.argmax(Y_test, axis = 1) != y_pred) & (np.
 argmax(Y_test, axis = 1) == 11) & (y_pred == 8))
misclassified_q_as_g = np.where((np.argmax(Y_test, axis = 1) != y_pred) & (np.
 argmax(Y_test, axis = 1) == 16) & (y_pred == 6))
misclassified_u_as_v = np.where((np.argmax(Y_test, axis = 1) != y_pred) & (np.
 argmax(Y_test, axis = 1) == 20) & (y_pred == 21))
misclassified_v_as_u = np.where((np.argmax(Y_test, axis = 1) != y_pred) & (np.
 argmax(Y_test, axis = 1) == 21) & (y_pred == 20))

miss_list = [misclassified_i_as_l[0], misclassified_g_as_q[0],
  misclassified_l_as_i[0], misclassified_q_as_g[0], misclassified_u_as_v[0],
  misclassified_v_as_u[0]]
labels = ['I as L ', 'G as Q ', 'L as I ', 'Q as G ', 'U as V ', 'V as U ']

fig, ax = plt.subplots(6, 7, sharex='col', sharey='row', figsize=(10, 10))
for i in range(6):
  for j in range(7):
    if j != 0:
      data = X_test[miss_list[i][j]].reshape((28, 28))
      ax[i, j].imshow(data, cmap='gray')

    if j == 0:
      ax[i, j].text(0, 0.5, labels[i], fontsize=18, horizontalalignment='left',
  verticalalignment='center', transform=ax[i, j].transAxes)
      ax[i, j].axis('off')
    ax[i, j].tick_params(left=False, labelleft=False, bottom = False, labelbottom
 = False)

"""## Version Printing"""

import sys
from kerastuner import __version__ as ktver
from scipy import __version__ as spver
from matplotlib import __version__ as mlpver

print(np.__version__)
print(pd.__version__)
print(pickle.format_version)
print(tf.__version__)
print(spver)
print(mlpver)
print(ktver)
print(sys.version)
```