

AutoClassMK: A public neural network for automatic 2D MK classification of normal stars in basic Python

C. Ian Short

Department of Astronomy & Physics and Institute for Computational Astrophysics, Saint
Mary's University, Halifax, NS, Canada, B3H 3C3

`ian.short@smu.ca`

Received _____; accepted _____

ABSTRACT

We present **AutoClassMK**, a simple, fully-connected, five-layer double-headed neural network written entirely in Python and Numpy that classifies *normal* stellar spectra conforming to the **libr18** MK atlas of Gray & Corbally (2014) in the 2D MK classification system with a high degree of precision and recall. **AutoClassMK** has the distinction of having transparent basic code with no calls to specialized libraries. In this paper we take care to explicitly describe in detail the ideas and operations that enable the network. Training AutoClassMK required us to develop large, noisy artificial training and test sets by augmenting the **libr18** and **libr18.27** MK atlases and to simplify the luminosity classification so that every combination of spectral- and luminosity-class is represented in the training set. We then test the network’s ability to predict the MK spectral type of noisy augmentations of spectra in the **libr18.225** MK atlas. We then implemented the same architecture in PyTorch to gain further insight and to enable execution on CUDA GPU’s. All codes and the training and test sets are available from the OpenStars [www](http://www.ap.smu.ca/OpenStars) site: www.ap.smu.ca/OpenStars.

Subject headings: stars: general, methods: numerical

1. Introduction

A star’s Morgan-Keenan (MK) spectral type is ideally an independent parameter that is independently determined so that any correlations with other observable quantities or fitted modeling quantities can be subject to on-going reinvestigation. Therefore, the classification is ideally strictly empirical and is based entirely on a program spectrum that conforms with the MK standard: A wavelength range ($\Delta\lambda$) of ~ 3900 to ~ 4900 Å , a spectral resolving power (R) of ~ 2000 , a reciprocal linear dispersion ($\frac{\Delta\lambda}{\Delta x}$) of ~ 67 Å mm⁻¹ (and, originally, a glass plate detector with the Kodak OJ III photographic emulsion and a spectral image with a widening of ~ 1 mm.) The classification criterion is strictly morphological: Overall qualitative visual pattern matching against an MK atlas such as that of Abt *et al.* (1968), or, currently, a digital MK atlas such as `libr18` (Gray & Corbally 2014). Therefore, automation of MK classification can benefit from the application of neural networks (NN’s), which can model an arbitrarily complex function relating sample content to probabilities corresponding to sample class. For ”normal” stars, the basic system is $2D$ and the complete MK spectral type consists of the spectral class (SC, including numerical subclass) and the luminosity class (LC).

The purpose of **AutoClassMK** is to de-mystify and publicize the basic methodology of simple NN’s, especially back-propagation, given their increasing importance to astronomy yet their opaque nature. A distinguishing property of **AutoClassMK** is that it has been developed entirely in basic Python and Numpy and every step is transparent for inspection and modification and the code could be a suitable starting point for a student project. To that end, **AutoClassMK** is available on the OpenStars [www](http://www.ap.smu.ca/OpenStars) site (www.ap.smu.ca/OpenStars).

NN’s that automatically classify stellar spectra have already been developed dating back to the pioneering work of Weaver & Torres-Dodgen (1997) and Bailer-Jones, Irwin & von

Hippel (1998) who developed and investigated simple NN’s like the one we present here. More recently, more sophisticated NN’s incorporating convolution (CNN’s) and principal component analysis (PCA) among other ideas that can work with large objective prism survey data and distinguish peculiar stars, among other things, have been presented (see, for example, Han, Kang & Jung (2026), Li (2025) (MCA-Net), Fu *et al.* (2024) (SFNet), and Wu *et al.* (2024)).

In Section 2 we describe our large augmented training and test sets, in Section 3 we describe the NN architecture and model, in Section 4 we present statistics describing how well the NN predicts the test set during training and an independent set of samples after the NN has been trained, and in Section 5 we describe a PyTorch implementation.

2. Training and test sets

We produce an adequate training set by augmenting the libr18 atlas. We begin by clipping the red end of the spectra to eliminate edge effects that might complicate the training. We then simplify the luminosity classification by re-labelling LC II stars as LC III and LC IV stars as LC V. LC Ia and Ib stars are re-labeled as LC I. This ensures that every combination of SC and LC is represented by at least one sample so that there are no empty cells in the $2D$ training set that might undermine the training.

For each spectrum in the libr18 atlas (*ie. primary*) spectrum, we produce 300 noisy, randomly erroneously calibrated variations corresponding to 300 different *normal* stars of identical MK type.

1. We assume a Gaussian distribution of random zeroth- and first-order horizontal registration errors corresponding to random small variations in zero-point positioning

and linear dispersion with values of σ of 0.030 and 2.0×10^{-5} equivalent \AA , respectively. We are effectively encoding wavelength, λ , with pixel number, so we do not explicitly account for errors in λ scale calibration.

2. We assume a Gaussian distribution of random errors in the Doppler-shift to the lab frame with a σ value of 1.0 km s^{-1} , which is additive with the horizontal registration variation.
3. We assume a Gaussian distribution of random errors in the zeroth-, first-, and second-order polynomial fitting coefficients used for continuum rectification, with σ values of 2×10^{-3} , 5×10^{-3} , and 3×10^{-3} , respectively, which gives rise to vertical variation due to a distribution of overall normalization levels and residual slopes and curvatures.
4. We assume a Gaussian distribution with a σ value of 0.3 pixels for variation among the standard deviation of Gaussian convolution kernels used to broaden the primary spectrum to account for both a random distribution of macroturbulent velocity dispersion, ξ , and rotational $v \sin i$ values. This will disproportionately affect weaker spectral features. We take a Gaussian to be a good approximation for the rotational broadening kernel for slowly rotating stars.
5. We assume that the primary spectra are brightness-limited and their noise is that of Poisson statistics. We assume a gain of 5000 counts and use the Numpy routine `poisson` to generate 300 noise spectra that we add to the primary spectrum, increasing the vertical variation. This is equivalent to assuming a signal-to-noise of ~ 70 , which is a significant underestimate. However, we deliberately overestimate the noise so as to train the NN to recognize noisy spectra.

We augment each of the 122 spectra in our pruned version of the `libr18` atlas to produce a

training set of 36600 spectra. These are then stored in random order so that a sequence of records corresponds to a random selection of spectral types. Fig. 1 shows the original spectrum and the 300 augmented spectra for MK type O9 V.

To assess the state of training (convergence) of the NN after each iteration, we require a *test* set of comparable samples that are independent of the training set. We generate a test set by following the data augmentation procedure described above to generate 30 variations of each of the 97 spectra in the libr18_27 atlas (Gray & Corbally 2014), giving us a test set of 2910 spectra.

3. The network

We have developed the simplest type of NN: A *fully connected* (FC, or *dense-layer*) network (a *perceptron*) in which the value of each element in a $1D$ array representing a sample at a particular layer of the network is generally determined by the value of *all* the elements in an array representing the sample at the previous layer. The equivalent description in the language of NN's is that each "neuron" in a given layer is affected by all the neurons in the previous layer.

3.1. Architecture

Our network has three hidden layers for a total of five main layers. The input layer has 1600 neurons, one per input pixel in the digitized training and test spectra. Each training and test sample consists of a set of normalized fluxes *vs* pixel number. This is tantamount to encoding physical real-number λ values with whole numbers.

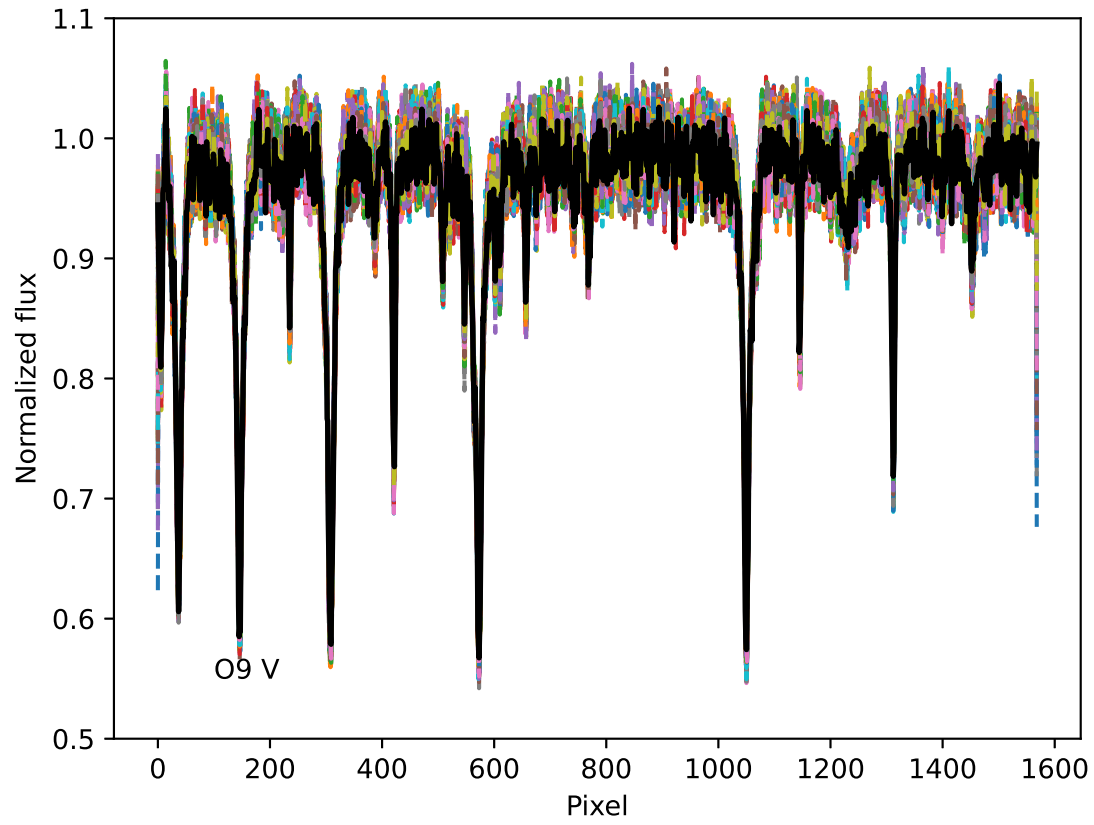


Fig. 1.— Original spectrum (thick black line) of class O9 V from the libr18 atlas (Gray & Corbally 2014) and 300 noisy, randomly mis-calibrated variations (dashed colored lines).

The output layer is bifurcated with two parallel linear transformations from the second-to-last layer, so that the network is *double-headed* for 2D classification. The SC head has 31 neurons and the LC head has three neurons, corresponding to the 31 predictable spectral classes and the three predictable luminosity in our training set. Our network has a *tapered* architecture and for the three hidden layers we have chosen 768, 256, and 64 for the numbers of neurons.

During training of the NN, a 1D (vectorized) representation of a stellar spectrum (a "sample") enters the input layer, and the sample moves forward through the NN by way of linear transformations that reduce the number of elements representing that sample. When passing from layer k with m neurons to layer $k + 1$ with n neurons, the sample is transformed from being represented with vector \vec{z}_k of length m to vector \vec{z}_{k+1} of length n by way of the linear transformation

$$\vec{z}_{k+1} = \bar{W}_k \cdot \vec{z}_k + \vec{b}_k \tag{1}$$

where \bar{W}_k and \vec{b}_k are the transformation matrix and the bias vector for a linear transformation *out of* layer k . With these transformations alone, the network can numerically model with a *linear* approximation a function that relates the sample contents (spectral features) to probabilities corresponding to classes (SC's and LC's). Training the network involves iteratively converging the values of \bar{W}_k and \vec{b}_k for all layers, k . The values at either final layer (head), \vec{z}_K , are referred to as *logits*, where K is the total number of layers.

3.2. Activation function

To allow the NN to model the *non*-linear relationship between the sample content (spectral features) and the probabilities of each spectral type, we operate on the transformed sample at each layer of $k < K$ with a non-linear activation function, f_A so that

$$\vec{a}_k = f_A(\vec{z}_k), \quad (2)$$

where \vec{a}_k is the *activated* representation of the sample at layer k . We have experimented with two common activation functions, Rectified Linear Units ($\text{RELU}(x)$) and Gaussian Error Linear Units ($\text{GELU}(x)$), where x is a pre-activated value. We also require the derivative, $\frac{df_A}{dx}$ for back-propagation. The RELU function is

$$f_A(x) = x, \text{ if } x > 0 \quad (3)$$

$$= 0, \text{ if } x < 0 \quad (4)$$

We avoid computing the Gaussian Error function by adopting the common approximation for the GELU function,

$$f_A(x) \approx 0.5x\{1 + \tanh[\sqrt{\frac{2}{\pi}}(x + cx^3)]\} \quad (5)$$

where c is 0.044715. The advantage of RELU activation is that it can be computed very quickly. However, the piecewise nature of RELU is known to introduce unphysical jaggedness in the relationship between $\Delta\bar{W}_k$ and $\Delta\vec{z}_K$. Both the RELU and GELU activation functions and their derivatives are shown in Fig. 2.

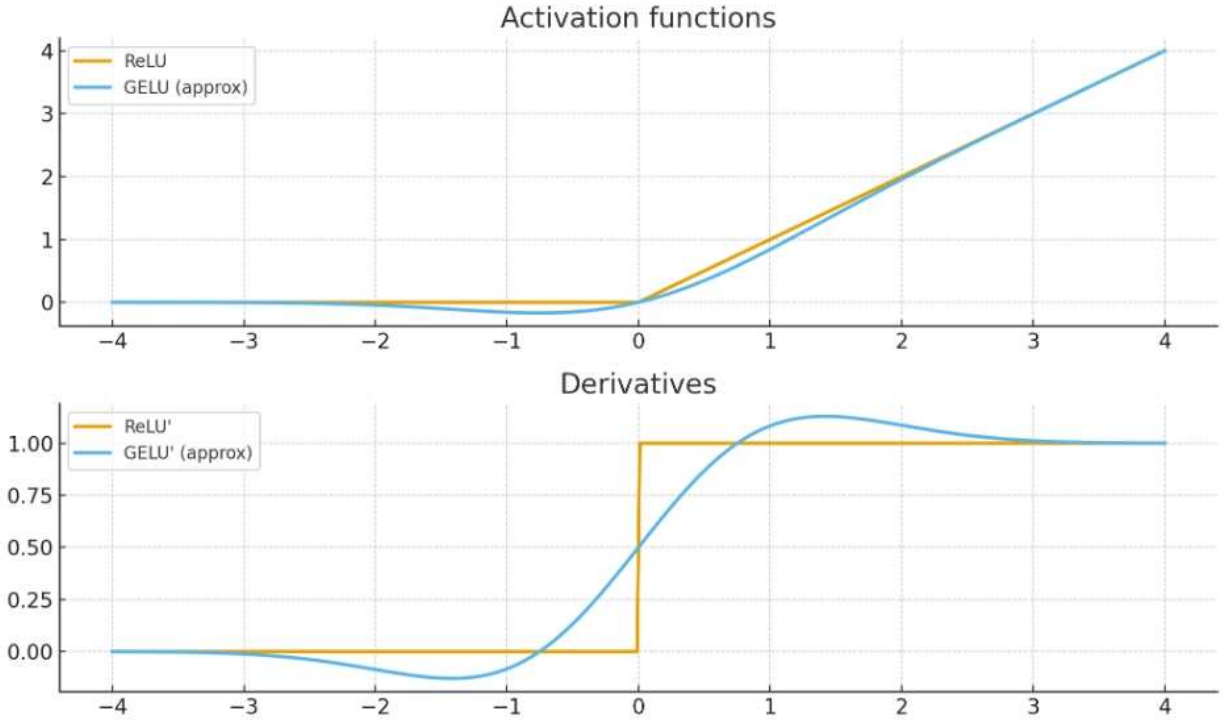


Fig. 2.— Post- *vs* pre-activated element value for the Gaussian Error Linear Units (GELU) activation function, f_A , (blue) used in our NN. Also shown for reference is the simpler Rectified Linear Units (ReLU) activation function (orange). (Credit: ChatGPT (OpenAI))

3.3. Probability distributions and loss

For each sample, the raw logits, \vec{z}_K , at the final layer, K , at either head are normalized among themselves by evaluating $z'_i = z_{K,i} - z_{K,\max}$ to mitigate against numerical instability. Then, for each sample, the vector of normalized logits, \vec{z}_K , is converted into a vector of probabilities, \vec{p} , with the **SOFTMAX** function

$$\vec{p} = \text{SOFTMAX}(z'_i) = \frac{\exp z'_i}{\sum_j \exp z'_j}. \quad (6)$$

The **SOFTMAX**(z'_i) function is equivalent to the Boltzmann distribution for excitation equilibrium where z'_i is the excitation energy relative to the ground state ($z_{K,\max}$) and $\sum_j \exp z'_j$ is the partition function. The **SOFTMAX**(z'_i) values are always positive and will sum to unity.

The sample labels for the true SC and LC are numerically encoded as label vectors, \vec{l} , with the *one-hot* encoding scheme, where a one-hot-encoded label, \vec{l} , is the result of operating on the true probability distribution, \vec{z}_{true} , with the **SOFTMAX** function, as

$$\vec{l} = \text{SOFTMAX}(\vec{z}_{\text{true}}) \quad (7)$$

This encodes the true SC or LC class label as a vector that is equal to zero everywhere except equal to one in the element corresponding to the true class. For example, in our simplified luminosity classification, if the true LC is I then the corresponding one-hot-encoded label is the vector $\vec{l} = [1, 0, 0]$ *ie.* the sample is an LC I star with probability one and an LC III or V star with probability zero.

We implicitly evaluate the average prediction error for each batch of samples after each training iteration with the cross-entropy (CE) loss, L , where, for a batch of N samples,

$$L = - \sum^N \vec{l} \cdot \ln \vec{p} \quad (8)$$

The advantage of one-hot encoding is that in a vectorized language like Numpy, the terms in Eq. 8 can be calculated simply with a vector operator. The CE loss, L , is defined so that each sample’s contribution to its batch’s L value at that iteration step depends on the probability distribution, \vec{p} , and, thus, the confidence of the current prediction, as well as the accuracy of the prediction. One can show that the definition of the **SOFTMAX** and L functions is such that

$$\frac{dL}{d\vec{z}} = \vec{p} - \vec{l} \quad (9)$$

where the gradient $\frac{dL}{d\vec{z}}$ is needed for back-propagation and can be evaluated simply. The value, L , is not used explicitly in back-propagation, but its definition is implicit because we use Eq. 9. We compute L for reporting purposes.

3.4. Forward pass and back-propagation

Initialization The matrices, \bar{W}_k , are initialized with a Gaussian distribution of random values centered on zero with a σ value of 0.1 for all layers, k . The bias vectors, \vec{b}_k are initialized to $\vec{0}$ for all k .

For each layer, k , we implement the transformation

$$\vec{z}_{k+1} = \bar{W}_k \cdot \vec{z}_k + \vec{b}_k \quad (10)$$

with Numpy’s vectorized dot-product operator. If k is equal to 1 then the operand, \vec{z}_1 , is the input sample, and if k is equal to K then the output, \vec{z}_K , is the final vector of raw logits at either of the two output heads. For the all k values of $k < K$, we activate the current representation of the sample by implementing

$$\vec{a}_k = f_A(\vec{z}_k) \quad (11)$$

For the layer $k = K - 1$ we implement two parallel transformations for the SC and LC heads.

3.4.1. Back-propagation

The batch-averaged error for the current predictions is

$$\delta \vec{z}_K = \frac{\vec{p} - \vec{l}}{N_{\text{samples}}} \quad (12)$$

Then, the corresponding errors in the matrix \bar{W}_K and the vector \vec{b}_K are

$$\delta \bar{W}_K = \vec{a}_{K-1}^T \cdot \delta \vec{z}_K \quad (13)$$

$$\delta \vec{b}_K = \sum^{N_{\text{samples}}} \delta \vec{z}_K \quad (14)$$

Eqs. 12 through 14 are executed twice, once for each of the SC and LC heads. The errors from the two heads are combined as

$$\vec{\delta a}_{K-1} = \vec{\delta z}_{K,SC} \cdot \bar{W}_{K,SC}^T + \vec{\delta z}_{K,LC} \cdot \bar{W}_{K,LC}^T \quad (15)$$

Subsequently, for all layers going backward through the network to decreasing k we have

$$\vec{\delta a}_{k-1} = \vec{\delta z}_k \cdot \bar{W}_k^T \quad (16)$$

$$\vec{\delta z}_k = \vec{\delta a}_k \cdot \frac{df_A(\vec{z}_k)}{dz} \quad (17)$$

$$\delta \bar{W}_k = \bar{a}_{k-1}^T \cdot \vec{\delta z}_k \quad (18)$$

$$\vec{\delta b}_k = \sum^{N_{\text{samples}}} \vec{\delta z}_k \quad (19)$$

For the special case of $k = 2$ we have

$$\delta \bar{W}_2 = \bar{z}_1^T \cdot \vec{\delta z}_2 \quad (20)$$

where \bar{z}_1 is the input sample.

3.5. Optimization step and learning rate

For each batch of samples for which we calculate batch-average $\delta \bar{W}_k$ and $\vec{\delta b}_k$ values, we perform an n^{th} update, or optimization step, for all layers, k , as

$$\bar{W}_k^{n+1} = \bar{W}_k^n - \lambda \delta \bar{W}_k \quad (21)$$

and

$$\vec{b}_k^{n+1} = \vec{b}_k^n - \lambda \delta \vec{b}_k \quad (22)$$

where λ is the *learning rate* and serves as a damping parameter to mitigate against oscillations that might prevent convergence or lead to divergence. Typical values of λ for 1D FC NN's range from 0.01 to 0.001. One can make use of a *learning rate schedule* to smooth convergence, where a typical schedule is one in which the value of λ is decreased by a factor of 2 every 50 outer iterations.

3.6. Mini-batch stochastic gradient descent (SGD)

The values of $\delta \bar{W}_k$ and $\delta \vec{b}_k$ used in the optimization step are averages computed for mini-batches of N_{samples} randomly drawn samples and reflect the average current prediction error for a random, approximately representative subset of the overall training set. If M is the total number of samples in the training set, then the number of mini-batches, N_{batch} , is M/N_{samples} . Iterative convergence is performed with a doubly nested loop in which the inner loop over mini-batches is executed N_{batch} times while the values of the \bar{W}_k and \vec{b}_k are cumulatively refined. The mini-batch loop is embedded in an outer loop over *epochs*, where each epoch the samples are randomly reshuffled among the mini-batches. If there are N_{epoch} epochs then the values of the \bar{W}_k 's and \vec{b}_k 's are cumulatively refined $N_{\text{epoch}} \times N_{\text{batch}}$ times, learning from $N_{\text{epoch}} \times N_{\text{batch}}$ unique randomly drawn mini-batches. Training with mini-batches rather than the overall training set greatly increases the number of optimization steps that occur for a given amount of processing.

The value of N_{samples} is chosen to be just small enough that a given mini-batch has a distribution of spectral types that is only approximately representative of the overall training set. This allows the iterative convergence to proceed stochastically, and provides noisy dithering that mitigates against the solution becoming trapped in a local shallow minimum.

4. Results

Figs. 3 and 4 show the CE loss, L , and the accuracy (**acc**), respectively, *vs* iteration number for our SC and LC training. After a relatively rapid early decrease in L , and corresponding increase in **acc**, during the first 50 - 100 iterations, the rate of convergence slows and asymptotically approaches limiting values of L of ~ 0.018 for SC and ~ 0.002 for LC, and limiting values of **acc** of ~ 0.88 for SC and ~ 0.97 for LC, within ~ 100 iterations. The convergence is non-monotonic on the scale of a few iterations, and our experience is that a learning-rate schedule in which the value of λ is decreased by a factor of 2 every 50 outer iterations can smooth the convergence, but at a cost of significantly increasing the number of iterations required.

In what follows N_{TP} , N_{FP} , and N_{FN} denote the numbers of true positive (TP), false positive (FP), and false negative (FN) classifications. Table 1 shows the **acc** value, the *precision* ($N_{\text{TP}}/(N_{\text{TP}} + N_{\text{FP}})$), the *recall* ($N_{\text{TP}}/(N_{\text{TP}} + N_{\text{FN}})$), and the *F1* score ($2 \times \text{precision} \times \text{recall} / (\text{precision} + \text{recall})$) computed with the `sklearn.metrics.classification_report` module for the SC and LC, evaluated using our *test* set of 2910 augmented noisy samples from the `libr18_27` atlas. Consistent with the behavior seen in Figs. 3 and 4, we see that the average *F1* score is greater for the LC (0.97) than it is for the SC (0.86).

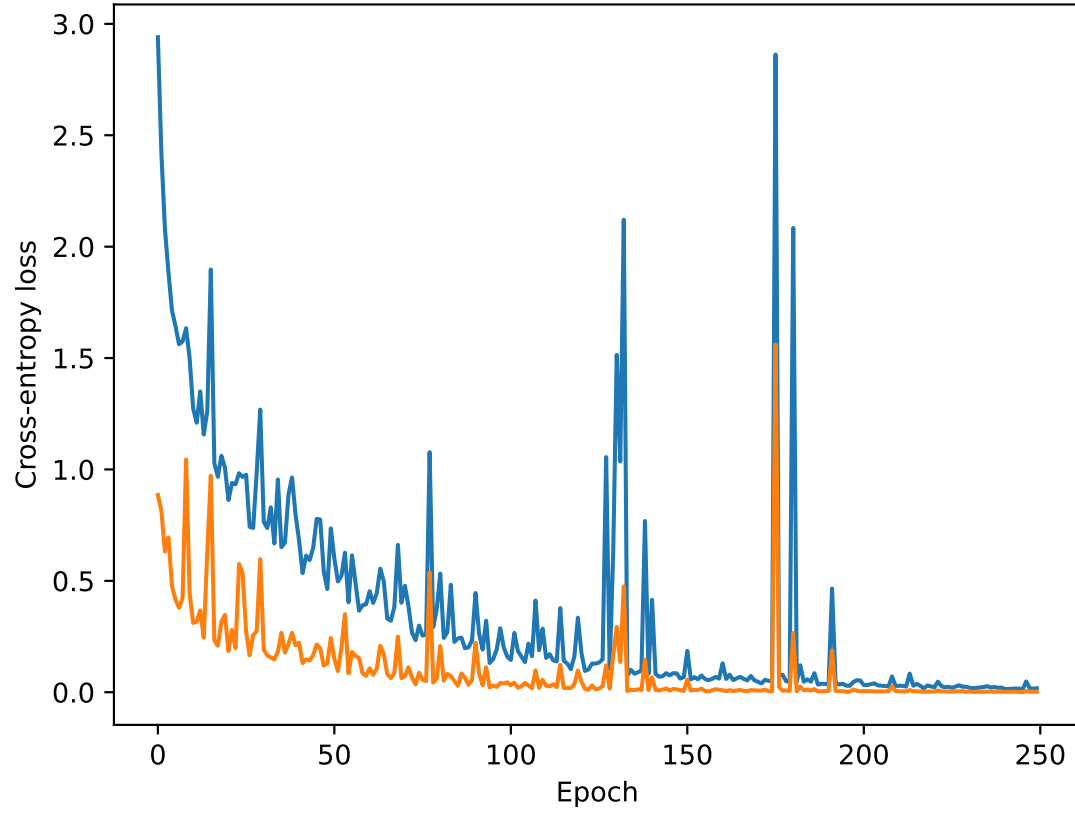


Fig. 3.— Mini-batch cross-entropy (CE) loss, L , *vs* outer iteration for our NN training with 250 outer iterations and a mini-batch size of 200 samples for the SC (blue) and LC (orange) classification.

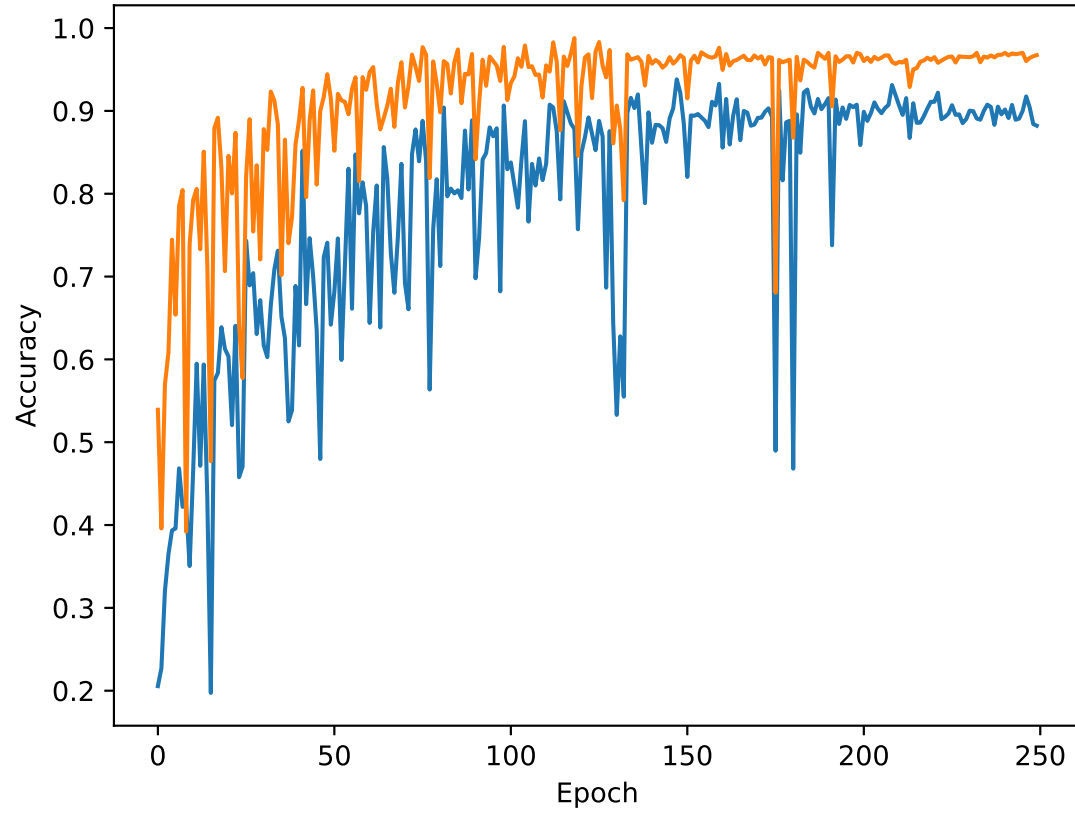


Fig. 4.— Same as Fig. 3 except that the dependent quantity is test-set prediction accuracy (acc).

Table 1: Training statistics.

Spectral Classification (SC)			
	Precision	Recall	F1
acc			0.88
Average	0.89	0.87	0.86
Luminosity Classification (LC)			
	Precision	Recall	F1
acc			0.97
Average	0.97	0.87	0.97

4.1. Prediction

Querying the trained NN (prediction) required executing the forward pass stage of the training described in Section 3. Samples must conform to the parameters of the `libr18` atlas in that the spectra must be *registered* such that the λ -range 3800-4600 Å spans 1600 pixels and samples must be continuum-rectified to unit continuum. We test our trained NN by using it to predict the SC and LC of augmented spectra taken from the `libr18_225` atlas (Gray & Corbally 2014). We follow the same augmentation procedure of Section 2 to produce three noisy, randomly mis-calibrated variations for each of the 85 spectra in our pruned version of `libr18_225`, for a total of 255 spectra in the set that our NN attempts to classify. Fig. 5 shows the predicted SC *vs* the true SC for each of the three LC values. Generally, the predicted SC is within one represented sub-class of the true SC.

Fig. 6 shows a 3×3 confusion matrix for the LC. We can see that the LC has excellent precision and recall for LC V and I. All 87 of the spectra with a true LC of V had correct LC predictions, and of the 81 spectra of true LC I, all but three had correct LC predictions.

Our NN had greater difficulty distinguishing LC III stars from LC V stars - of the 87 spectra of true LC III, 13 of them were predicted to be of LC V.

5. PyTorch implementation

To gain further insight into our NN and to facilitate execution on GPU’s equipped with CUDA, we implemented the same architecture in the PyTorch deep learning library (Paszke *et al.* 2019). The PyTorch implementation of our five-layer FC NN is shown in Table 2.

Table 2: AutoClassMK PyTorch architecture.

$$\begin{aligned}
 z_{\text{SC}}, z_{\text{LC}} &= \text{MODEL}(z_1) \\
 \text{CE}_{\text{SC}} &= \text{CELoss}(z_{\text{SC}}, z_{\text{SC}}^{\text{True}}) & \text{CE}_{\text{LC}} &= \text{CELoss}(z_{\text{LC}}, z_{\text{LC}}^{\text{True}}) \\
 \text{CE}_{\text{TOTAL}} &= \frac{(w_{\text{SC}}\text{CE}_{\text{SC}} + w_{\text{LC}}\text{CE}_{\text{LC}})}{(w_{\text{SC}} + w_{\text{LC}})} \\
 \text{SGD.ZERO_GRAD}(\text{MODEL}, \lambda) \\
 \text{CE}_{\text{TOTAL}}.\text{BACKWARDS}() \\
 \text{SGD.STEP}(\text{MODEL}, \lambda)
 \end{aligned}$$

where w_{SC} and w_{LC} are parameters that allow us to weight the relative importance of the errors of the SC and LC predictions in the training. Given that the NN has greater difficulty learning the SC- than the LC-dimension, we have found it advantageous to adopt $w_{\text{SC}} = 2$ and $w_{\text{LC}} = 1$ to force the NN to learn more from the SC prediction error during training. The `MODEL` is shown in Table 3.

The PyTorch version is implemented in a *device agnostic* way and will automatically run on a GPU if it detects a CUDA processor on the host. We have successfully run the PyTorch version on both a CPU and a GPU of the Digital Research Alliance of Canada (DRAC) machine nibi.

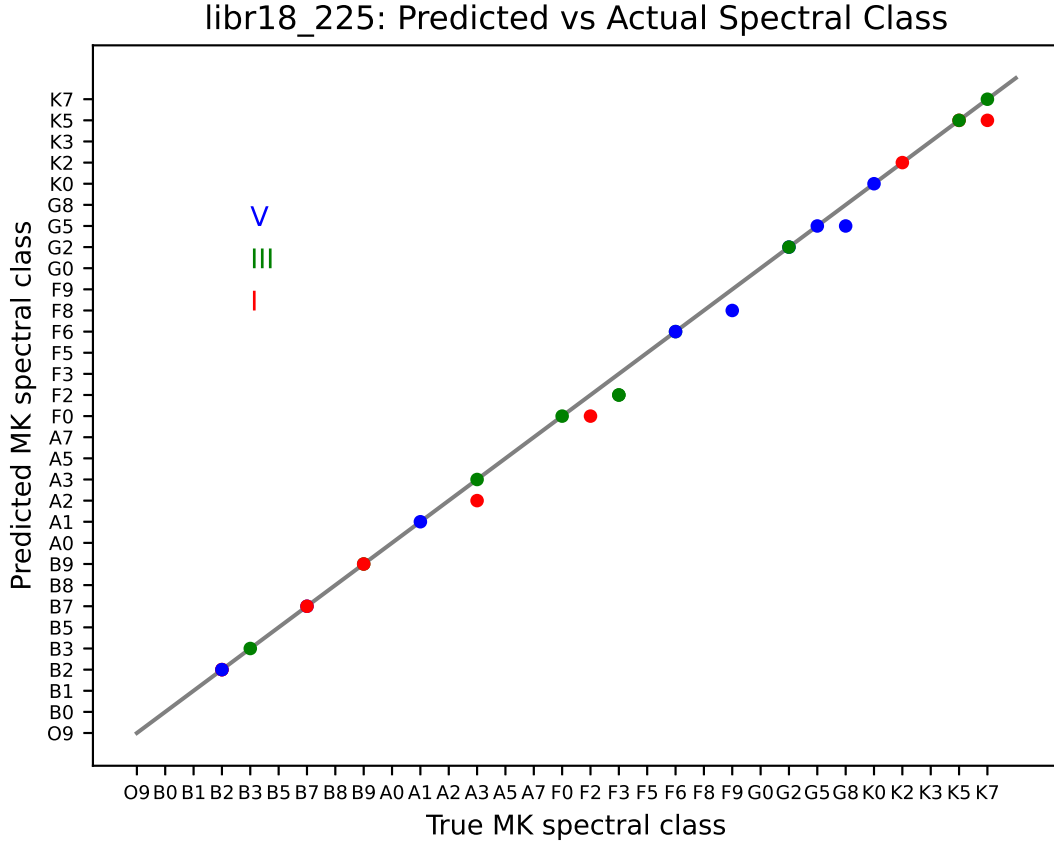


Fig. 5.— Predicted *vs* true SC for noisy augmentations of the libr18_225 (Gray & Corbally 2014) atlas for luminosity class V (blue), III (green), and I (red) stars. The gray line indicates the locus of prefect prediction.

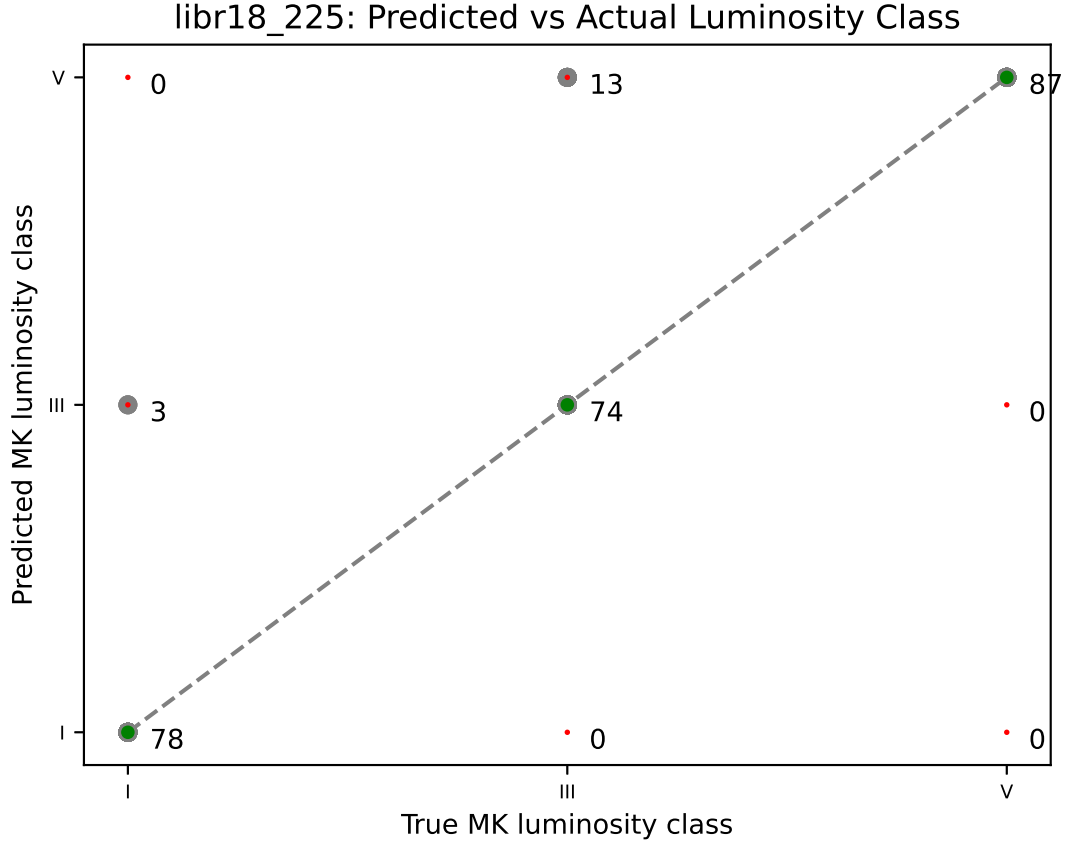


Fig. 6.— The 3×3 confusion matrix for predicted *vs* true LC for noisy augmentations of the libr18.225 (Gray & Corbally 2014) atlas. The locus of perfect prediction is the anti-diagonal. The annotations are the numbers of spectra with each combination of true and predicted LC.

Table 3: AutoClassMK PyTorch model (MODEL of Table 2).

$$\begin{aligned}
 z_1 &= \text{LINEAR}(1600, 768) \\
 a_1 &= \text{GELU}(z_1) \\
 z_2 &= \text{LINEAR}(768, 256) \\
 a_2 &= \text{GELU}(z_2) \\
 z_3 &= \text{LINEAR}(256, 64) \\
 a_3 &= \text{GELU}(z_3) \\
 z_{\text{SC}} &= \text{LINEAR}(64, 31) & z_{\text{LC}} &= \text{LINEAR}(64, 3) \\
 z_{\text{SC}} &= \text{SOFTMAX}(z_{\text{SC}}) & z_{\text{LC}} &= \text{SOFTMAX}(z_{\text{LC}})
 \end{aligned}$$

6. Future Work

AutoClassMK and the accompanying training and test sets provide ample opportunities for experiments that a student could do with nothing other than the basic Python and Numpy installations:

- Studying how training convergence and the precision and recall of the trained network depend on the noisiness, mis-calibration variance, and number of samples per class, of the training set and on the noisiness of the set for which the classes are being predicted.
- Studying the how training convergence depends on arbitrary parameters such as the numbers of neurons in the hidden layers, the learning rate, λ , and the size of the mini-batches, N_{samples} .
- More substantially, studying the effect of adding and removing hidden layers.

We used ChatGPT (OpenAI) for conceptual discussion and code prototyping. All final

implementations and scientific interpretations were performed by the author. Fig. 2 was produced by ChatGPT (Open AI). This research was enabled in part by support provided by ACENET and the Digital Research Alliance of Canada (DRAC, alliancecan.ca). Some data preprocessing and numerical calculations were performed using the NumPy library, available at <https://numpy.org>, in Python.

REFERENCES

- Abt, H. A, Meinel, A. B., Morgan, W. W., Tapscott, I. W., 1968, *An Atlas of Low Dispersion Grating Spectra*, Tuscon
- Bailer-Jones, C.A.L., Irwin, M. & von Hippel, T., 1998, MNRAS, 298, 361
- Fu, H., Liu, P., Qi, X., Mei, X., 2024, Research in Astronomy and Astrophysics, 24, 095023
- Gray, R. O. & Corbally, C. J., 2014, AJ, 147, 80
- Han, S., Kang, W. & Jung, J-H, 2026, Astronomy and Computing, 54, 101024
- Li, H., 2025, Experimental Astronomy, 60, 17
- Paszke, A., *et al.*, 2019, arXiv:1912.01703v1 (PyTorch)
- Weaver, Wm. B., Torres-Dodgen, A. V., 1997, ApJ, 487, 847
- Wu, J., He, Y., Wang, W., Qu, M., Jiang, B., Zhang, Y., 2024, AJ, 167, 260
-