Solutions and Notes to the Problems in: Hands-on Machine Learning with Scikit-Learn, Keras, & TensorFlow by Auélien Géron

John L. Weatherwax*

October 17, 2023

Introduction

Géron's book on machine learning is such a great book on the practical aspects of machine learning and I would recommend everyone who works in this area to read his book. I actually read the book twice once for each of the two editions. The second time, to challenge myself and gain a better understanding of the material I decided to work the questions given in the back of each chapter. This document represents my answers to these questions. Its my hope that other students of machine learning might find my answers helpful for self study.

At the end of most of the chapters Géron has provided a number of interesting computational problems. Many of these extend the concepts discussed in the given chapter and provide further details. Some of his problems are quite involved and to do these problems "from scratch" could requires a great deal of time. Unfortunately, I didn't feel that I had the time to work all the computational problems to the point where I would feel acceptable presenting them. Fortunately for many of them, Géron has provided us with notebooks that implement them in great detail. I think following along with his code is a good way to learn the additional material without such a large investment in time. That was the approach I took. If I had something interesting to say about his approach I will present it in these notes but often his code was so straightforward that no comments were needed. I would advise everyone to read Géron's solutions as they read the main text. For the problems that Géron did not work I attempted to work them myself. Python notebooks to my solutions can be found at my web site.

*wax@alum.mit.edu

Chapter 1 (The Machine Learning Landscape)

Exercise 1

Machine learning attempts to use data and a model on how variables in the data should be related to one-another to build predictive relationships between variables. Once this relationship is defined (and estimated) predictions can be made for variables of interest when presented with new data.

Exercise 2

The book discusses solutions where machine learning works well

- If the solutions require a lot of hand tuning.
- Very complex problems where a traditional method to approach the solution is not fully known.
- Environments where the input data can change.
- Building understanding on very complex problems that can have large amounts of data.

Exercise 3

A labeled set is a set of data where the desired output is known. Thus for these problem instances we know the "answer".

Exercise 4

The two most common supervised tasks are regression (predicting a real valued output) and classification (predicting a class label).

Exercise 5

Four common types of unsupervised learning tasks (mentioned in the book) include

- Clustering.

- Anomaly and novelty detection.
- Visualization and dimensionally reduction.
- Association rule learning.

Exercise 6

This robot would have to learn to adapt and not fall in the different environments. Given that it would have to adapt its behavior (learning "not" to do things when they resulted in it falling over). I would think that reinforcement learning would be the appropriate technique to use in this case.

Exercise 7

This seems like a use case for a clustering algorithm. Here we are looking for clusters of similar customers.

Exercise 8

Most projects of this type have a large corpus of spam emails and so we should use supervised learning to solve this problem.

Exercise 9

Typically "online" means learning that is done incrementally as opposed to batch learning. This may mean that an online system can adjusts its behavior as it is operating. Thus an online system may use data that was not present during development to change its future decisions. One way this could be done is by using this new data as new training samples.

Exercise 10

Out-of-core learning is a term used when the dataset is so large that it cannot fit into a single machines memory. Typically parallel processing must be done in some way to process this data and the individual results combined to produce the final learning algorithm.

Exercise 11

Local algorithms like the k-nearest neighbors algorithm needs a distance (similarity) metric to determine if points in the feature space are close to each other.

Exercise 12

A hyperparameter is a parameter that controls *how* learning is performed and a parameter is *what* is learned. Once learning is performed the model parameters are estimated. Changing the hyperparameter to a different value (and relearning) will result in a novel set of model parameters. Often hyperparameters are used to control the bias-variance trade-off required to prevent overfitting.

Exercise 13

Model based learning algorithms seek to find parameters that make the observed data consistent with the given model. They can find these parameters by optimization, maximizing things like the likelihood of the data given the model (and parameters). These algorithms then make predictions by evaluating their representative model on new input points using the estimated parameters.

Exercise 14

The book mentions four challenges that all users of machine learning will face or at least must be able to handle

- Not enough training data.
- Non-representative training data.
- Overfitting the training data (learning the noise).
- Underfitting the training data (not learning all of the signal).

Exercise 15

You are overfitting your training set i.e. learning the noise found in your training set. You can

- Regularize your learning algorithm i.e. select one with less variance.
- Reduce the number of features you are considering and/or fix known data errors.
- Get more representative training data.

Exercise 16

The test set is a set of data held out during development and that is used during the final stages of your algorithm development to estimate performance of the algorithm on new data. One would use this set only to report how your final algorithm would perform on novel data. If your unhappy with that performance (and you seek to be unbiased in your estimates) there is nothing you can do. Well, you could keep working on your algorithm but then to really determine its out-of-sample performance you would need to obtain a different test set.

Exercise 17

The validation set is used to select the hyperparameters. For a fixed set of hyperparameters the training set is used to estimate the model parameters. The performance of this algorithm (with the just estimated parameters) is then evaluated on the validation set. For each set of hyperparameters we now have an estimate of the performance on a "novel" dataset. Finally, we then select the hyperparameters that give the best validation set performance.

Exercise 18

If you tune the hyperparameters in the testing set you would select the hyperparameters that give the algorithm the greatest variance i.e. the greatest amount of power to fit the testing set. The resulting algorithm would not generalize well to new data as you are overfitting the test set.

Exercise 19

In k-fold cross validation one first breaks the data into k groups of roughly equal size. Each of these groups is called a fold. Then specifying one of the folds as the hold-out set we train our algorithm using all $k-1$ other folds as the data and then test on the held-out fold. If we do that for each of the k folds we get k estimates of the performance of our algorithm on data not seen during training. These can be used to give an estimate of the out-of-sample performance of our learning algorithm. This is better than a single validation set in that we now have k estimates of this same number and the average of them will have less variance than a single measurement (like what would be obtained if we only had the single validation set).

Chapter 2 (End-to-End Machine Learning Project)

Here I look at the exercises for this chapter. Note that these exercise are worked in notebooks presented by the book and thus one can get solutions there if desired. At the time of this writing the book's solutions can be found

`https://github.com/ageron/handson-ml2/blob/master/02_end_to_end_machine_learning_project.ipynb`

As these solutions are quite good instead of "reinventing the wheel" I read the provided notebooks/solutions, ran the resulting codes, and used them as a starting point for any further analysis (if I had time to do any). In these notes I'm simply providing comments on anything interesting I observed as I followed that procedure.

Exercise 1

We specify a relatively large parameter grid to search using the function `GridSearchCV`. Due to the large number of parameters running that code takes quite a bit of time to run. Currently the best RMSE error is from the `RandomForestRegressor` with a value of about 49000 for `max_features` of seven and `n_estimators` of 180. The cross-validated SVR model is much worse with a RMSE of around 70000.

We notice that when using the parameters given in the notebook the best result happens at the value of $C = 30000$ which is the largest one tested. To following the suggestions in the notebook we should remove the smaller values of C and test larger ones. Given that there is such a discrepancy between the the best SVR RMSE and the best RMSE from the `RandomForestRegressor` I assume that this change will not produced a SVR model that is better than the `RandomForestRegressor` and thus I choose not to implement it.

Exercise 2

We next try to fit the housing data using `RandomizedSearchCV` which takes "distributions" (or lists) over the parameters we want to test and then performs cross-validation over a fixed number of such parameter settings. This is quite nice in that we can very easily test as many parameter variations as we have time for.

Running the code there, we able to find parameter settings that work much better than we found with `GridSearchCV` getting a RMSE of around 55000 which is much closer to that obtained by the `RandomForestRegressor`. Note that the optimal value of C obtained is much larger than the 30000 tested in the previous exercise.

The use of the "reciprocal" distribution to find hyperparameters when you have very little knowledge of the value of the hyperparameters value is quite interesting.

Chapter 3 (Classification)

Here I look at the exercises for this chapter. Note that these exercise are worked in notebooks presented by the book and thus one can get solutions there if desired. At the time of this writing the book's solutions can be found

`https://github.com/ageron/handson-ml2/blob/master/03_classification.ipynb`

As these solutions are quite good instead of "reinventing the wheel" I read the provided notebooks/solutions, ran the resulting codes, and used them as a starting point for any further analysis (if I had time to do any). In these notes I'm simply providing comments on anything interesting I observed as I followed that procedure.

One of the exercises for this chapter (spam email classification) could be expanded into a chapter of a book given the level of detail in which Auélien's presents its solutions. It seemed very hard to improve much on his solution to this question!

Exercise 1

This is done in the accompanying notebook. Running the code took several hours for me. This was mostly due to the cost of the `GridSearchCV` call.

Exercise 2

This is done in the accompanying notebook. Running that code we see that by augmenting our data we get a classifier with a larger accuracy. Note that the `predict` method takes a very long time on this data. In this exercise the augmentations resulted in "valid" images and hence valid training samples. This makes me wonder if one has a machine learning problem with a lot of noise one might be able to improve on baseline results by adding noised samples from the original dataset. This seems like the procedure "bagging" but involves only modifying the training data and not producing multiple classifiers/predictors.

Exercise 3

Here Auélien presents some solutions with different performance measures (accuracies). He gets a jump in performance by using a `RandomForestClassifier`. Random forests are well known to be very good learning models. He then suggests a couple of preprocessing steps one could take to potentially produced more informative features.

Exercise 4

Here to do this exercise Auélien must load and parse the raw emails we are instructed to download. This alone seems to involve a great deal of work/code. Its great to see such a complete data pipeline for a somewhat complicated application. This "exercise" is really the start of a suite of tool for extracting features from actual emails as the application of the machine learning part is only a few lines of code once the preprocessing is done!

Chapter 4 (Training Models)

Exercise 1

If you have millions of features the naive implementation of linear regression using the normal equations would be too difficult computationally to perform. One could instead use a gradient decent technique (like stochastic gradient decent or mini-batch gradient decent) which requires only inner products and avoids the matrix inversion required in the normal equations.

Exercise 2

The minimization surface that represents the error obtained when using a model with different coefficients can have a high curvature if the features have very different scalings. This can cause fitting routines to have numerical difficulties finding the optimal parameters and/or take a very long time to converge. I think in general most algorithms will have some degree of difficulty if the feature scaling is too extreme. To work around this issue one can "standardize" the features (subtract the mean and divide by the standard deviation) before providing them to an upstream machine learning algorithm.

Exercise 3

Gradient decent cannot get stuck in a local minimum when training a logistic regression model as the error surface is convex and thus has only one minimum.

Exercise 4

Gradient decent algorithms have various parameters and if these are not set correctly the algorithm may not even converge. If the parameters are set correctly for convergence some gradient decent techniques like stochastic gradient decent will "oscillate" around the global minimum will will never converge to the exact minimum as specified by the solution to the normal equations. For linear regression and logistic regression which are convex problems the minimum selected for all of these techniques (when they converge) will be quite close and the resulting models very similar.

Exercise 5

If the validation error is increasing it means that the learning rate parameter η, is too large. You should decrease your stepsize parameter and run again.

Exercise 6

Due to the random nature of mini-batch gradient decent the algorithm is not guaranteed to decrease the MSE at every iteration (just on average). Thus you might want to run the algorithm for a bit longer and output the parameter setting that gave the smallest validation error form all mini-batch steps.

Exercise 7

Stochastic gradient decent will reach the vicinity of the minimum first but will oscillate around the minimum forever (unless you decrease the learning rate η). Only batch gradient decent will actually converge.

Exercise 8

This looks like a symptom of overfitting the training set. One solution would be to reduce the dimension of the polynomial or add a regularization penalty to the fit.

Exercise 9

I would say high bias as the model is not fitting the data very well (it is biased away from the true signal) and maybe underfitting the data. One would want to reduced the regularization parameter α i.e. make the model less regularized.

Exercise 10

- We would use ridge regression instead of linear regression if we had a large number of features and felt we would overfit our training data if we didn't have any regularization.
- You would prefer lasso rather than ridge regularization if you wanted the ability to drop features as the lasso has an ℓ_1 penalty which performs some amount of feature selection automatically when fitting the model with this penalty. If you expect that only a few features actually matter you should prefer the lasso regression otherwise prefer ridge regression.

- The elastic net is a fitting procedure that has two hyperparameters (one for the ℓ_1 constraint and another for the ℓ_2 constraint). Fitting an elastic net model can provide some of the benefit of each of the two types of models.

Exercise 11

Notice that these four states are not mutually exclusive i.e. one could be outdoors during the day or night. Because of this one should two logistic regression classifiers rather than one softmax regression classifier. The softmax classifier would perform better with states that are mutually exclusive.

Exercise 12

This would be implemented in much the same way as the early stopping using stochastic gradient decent but using vector operations to compute the gradient of the cross-entropy with respect to the parameters (rather than compute a single step with a call to `SGDRegressor`). Again the computational solutions provided by Auélien are so good that for my purposes it is enough to read and understand the code he provides. The solutions he provides don't *actually* implement early stopping in the learning loops but that is easy enough to add by following the discussion in the book on that topic.

It is instructive to consider some simplifications that can be made when there are only two classes and we use softmax regression. In using the softmax regression each class is modeled with a linear logit as

$$s_k(x) = \beta_{k,0} + \beta_k^T x \,.$$

These logits are combined to produce a probability as

$$P_k = \frac{e^{s_k(x)}}{\sum_{j=1}^{k} e^{s_j(x)}} \,.$$

Now if there are only two classes we have $s_k(x)$ for $k \in \{1, 2\}$ and

$$P_1 = \frac{e^{s_1(x)}}{e^{s_1(x)} + e^{s_2(x)}} = \frac{1}{1 + e^{s_1(x) - s_2(x)}}$$

and

$$P_2 = \frac{e^{s_2(x)}}{e^{s_1(x)} + e^{s_2(x)}} = \frac{e^{s_2(x) - s_1(x)}}{1 + e^{s_1(x) - s_2(x)}} \,.$$

Thus in this formulation since

$$s_1(x) - s_2(x) = \beta_{1,0} + \beta_1^T x - (\beta_{2,0} + \beta_2^T x) = \beta_{1,0} - \beta_{2,0} + (\beta_1 - \beta_2)^T x = \tilde{\beta}_0 + \tilde{\beta}^T x \,.$$

we have

$$P_1 = \frac{1}{1 + e^{\tilde{\beta}_0 + \tilde{\beta} x}} \,,$$

with a similar expression for P_2. This has changed our problem from one with "four" unknowns

$$\beta_{1,0}\,,\beta_1\,,\beta_{2,0}\,,\beta_2\,,$$

to one with only "two"

$$\tilde{\beta}_0\,,\tilde{\beta}_1\,.$$

Some software packages only understand one of these parameterization and its helpful to be able to translate between the two representations.

Chapter 5 (Support Vector Machines)

Exercise 1

The idea of a support vector machine is to place a hyperplane (the decision boundary) in between the training data in such a way that it separates the two classes "maximally". That is we desire the distance between the data in each class and the hyperplane to be as large as it can be. From the decision boundary the algorithm would measures how far away the nearest point in each class is from the decision boundary. This width is called the "margin" and if the classes are linearly separable is a well defined thing. In that case the margin is called "hard". In non-linearly separable classes the margin is called "soft" due to the fact that there maybe misclassified points.

Exercise 2

After training a support vector machine the final prediction is made based on the test points location with respect to the learned hyperplane. Thus if we seek to classify x then we compute

$$\hat{y} = \begin{cases} 0 & \hat{w}^T x + \hat{b} < 0 \\ 1 & \hat{w}^T x + \hat{b} \geq 0 \end{cases} .$$

Here $\hat{w}$ and $\hat{b}$ are learned from the training procedure. In fact a formula for $\hat{w}$ can be given by

$$\hat{w} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} x^{(i)} .$$

Here $t^{(i)} \in \{0, 1\}$ are the class "labels" for the ith training instance, $x^{(i)}$ is the ith feature vector, and $\hat{\alpha}^{(i)}$ are the output from the dual form of the linear SVM. In that optimization many of $\hat{\alpha}^{(i)}$ are zero. The features $x^{(i)}$ with $\hat{\alpha}^{(i)} > 0$ are the support vectors. Any vector that is not a support vector can be removed and it will have no effect on the decision boundary.

Exercise 3

As the optimization problem in SVM involves maximizing a "distance" in the features space it is important to have each feature have the same scaling. For example, if one feature is gross domestic product (GDP) measured in billions of dollars and another feature is a person's height measured in feet, then any distance measured between these two vectors will be dominated by the GDP numbers. Scaling each feature by using `StandardScalar` in `sklearn.preprocessing` will turn each feature into a z-score which will allow feature vectors to be compared with respect to distances.

Exercise 4

The SVM can output the distance x is from the learned hyperplane i.e. $|\hat{w}^T x + \hat{b}|$ which could be used as a confidence. There is no natural notion of a probability in SVM classifiers but one could be learned by using these confidence scores as input to a logistic regression classifier. Setting `probability=True` when creating a SVM in Scikit-Learn will perform these steps automatically and allow probability to be calculated in this way.

Exercise 5

We are told that our problem has $m = O(10^6)$ and $n = O(10^2)$ so we have $m \gg n$ i.e. more training instances than number of features. In this case the direct problem formulation is easier (computationally) to solve.

Exercise 6

The Gaussian RBF takes the form

$$\phi_\gamma(x, l) = \exp\left\{-\gamma||x - l||^2\right\} .$$

Thinking of the parameter γ as an "inverse variance" we see that when $\gamma \gg 1$ (so that $\frac{1}{\gamma} \ll 1$) that the ϕ functions are highly peaked around the l vector and the classifier would have high variance with low bias. If $\gamma \ll 1$ (so that $\frac{1}{\gamma} \gg 1$) then the ϕ function are more diffuse around the l vector and the classifier would have low variance and high bias. Since we are told that our classifier is underfitting we should make γ larger (to move it into the low bias regime).

In Scikit-Learn the C parameter is used to regularize the solution where the strength of the regularization is proportional to $\frac{1}{C}$. As we are underfitting we are too regularized and we should increase the value of C (to make $\frac{1}{C}$ smaller).

Exercise 7

Recall that in the notation of the textbook m is the number of training samples and n is the number of features.

The soft margin problem for the linear SVM classifier is to *minimize* (over the variables w, b, and ζ) the objective

$$\frac{1}{2} w^T w + C \sum_{i=1}^{m} \zeta^{(i)} . \tag{1}$$

subject to the constraints

$$t^{(i)}(w^T x^{(i)} + b) \geq 1 - \zeta^{(i)}\,, \tag{2}$$

and

$$\zeta^{(i)} \geq 0\,, \tag{3}$$

for $i = 1, 2, \ldots, m$.

Now to use an off-the-shelf QP solver we will define the vector of unknowns p to be

$$p = \begin{bmatrix} w \\ b \\ \zeta \end{bmatrix}\,.$$

Each of these vectors is vertically concatenated and thus the size of vector is $n_p = n+1+m = m+n+1$. Next H is a block diagonal matrix with blocks

- $I_{n\times n}$ (the $n \times n$ identity matrix)
- $0_{1\times 1}$ (a 1×1 zero matrix)
- $0_{m\times m}$ (a $m \times m$ zero matrix)

This means that H is a matrix of size $(m+n+1) \times (m+n+1)$. Next note that the vector f is a "block" column vector with n zeros, one zero, and n C's.

To determine the A in the off-the-shelf constraint $Ap \leq b$ we write Equation 2 as

$$-t^{(i)}(w^T x^{(i)} + b) \leq \zeta^{(i)} - 1\,,$$

or

$$-t^{(i)} w^T x^{(i)} - t^{(i)} b - \zeta^{(i)} \leq -1\,, \tag{4}$$

for $1 \leq i \leq m$. Then we write Equation 3 as

$$-\zeta^{(i)} \leq 0\,, \tag{5}$$

for $1 \leq i \leq m$.

From these expressions we see that the matrix A is of size $2m \times (m+n+1)$. From Equation 4 we see that first $1 \leq i \leq m$ rows of A have the ith row taking the block form of

$$[-t^{(i)} {x^{(i)}}^T,\, -t^{(i)},\, -I_{(i)}]\,.$$

Here $I_{(i)}$ is a row vector of zeros (of size $1 \times m$) except for a one in the ith spot. The second set of m rows look like the first set but with the first two parts zero or

$$[0_{1\times n},\, 0_{1\times 1},\, -I_{(i)}]\,.$$

Finally, the column vector b has m negative ones followed by m zeros.

Exercise 8

Here I again simply followed Auélien's excellent solutions. This exercise seemed mostly aimed at exercising the functionality of `LinearSVC`, `SVC(kernel='linear')`, and `SGDClassifier`.

Exercise 9

Here I again followed Auélien's solutions. The key idea I took from this exercise is that one can do model selection on a small subset of data (so that the grid searches are much faster to execute than they would be on the full dataset) and then use the best performing models as "initial guesses" to be used for retraining on the full dataset. This initial training subset can help eliminate very poor hyperparameters from being considered further and greatly facilitate getting to a good set of hyperparameters in a shorter period of time. This idea seems to have other possible generalizations.

Exercise 10

Here I simply followed Auélien's solutions. We see that a `SVR` using radial basis functions can cut the error of a `LinearSVR` by about 50% which is a significant improvement.

Chapter 6 (Decision Trees)

Exercise 1

If each sample ends up at a leaf then there will need to be $\log_2(m)$ levels to this tree. When $m = 10^6$ this number is 19.93157.

Exercise 2

At a high level, the splitting we are doing at each parent node is to make the two child nodes more pure the impurity should shrink (i.e. *lower*) as we progress down through the tree. The impurity of the child notes will generally be lower but is not required to be lower than that of the parent node. This is because, if during training, a split is chosen to take the parent into a "small" and a "large" child node then it is possible for a smaller child to have a larger impurity than the parent if the corresponding larger child has its impurity smaller smaller than the parent node. The first edition of this text gives a nice toy example where this can happen.

Exercise 3

Yes. This will make the trees shallower and less dependent on the data thus helping to reduce overfitting. The shallowest tree is one with a depth of one and is called a "stump". By reducing the parameter `max_depth` you are constraining the fitting algorithm which is how one combats overfitting.

Exercise 4

No. Scaling the input features has no effect on the performance of decision trees. This is a benefit of using CART decision trees in that less initial work needs to be done to get a working model.

Exercise 5

From the book the training time of a CART tree is $O(nm\log(m))$ where n is the number of features and m is the number of samples. When $m = 10^6$ we are told that this expression approximately equals one hour. If now m increases by a factor of ten out training time would be (dropping the O notation for now)

$$n(10m)\log(10m) = 10(nm\log(m) + nm\log(10)) \approx 10nm\log(m)\,.$$

Thus it will take about ten times as long or ten hours.

Exercise 6

From the book, setting the parameter `presort=True` will speed up training for *small* datasets and will dramatically slow down the training of large datasets. Small here seems to be $m \approx O(10^3)$. The value of m in this exercise is too large for `presort` to help.

Exercise 7-8

Here I simply followed Auélien's provided solutions as they are an excellent learning tool.

Chapter 7 (Ensemble Learning and Random Forests)

Exercise 1

We can understand the improvement we might obtain if the errors made by each classifier are independent. If we assume a binary classification problem and combine the five classifiers using majority voting we will classify a class as "one" if three or more of the classifiers vote for "one". An error will then take place if three or more classifiers vote "zero". From the problem each classifier makes an error with a probability of 0.05. Thus the ensemble will make an error if three or more classifiers do. This will happen with a probability of

$$\binom{5}{3}e^3(1-e)^2 + \binom{5}{4}e^4(1-e)^1 + \binom{5}{5}e^5(1-e)^0 .$$

Here e is the error probability $e = 0.05$. If we evaluate this we get 0.001158125 an error much smaller than any individual classifier. This result is optimistic because the classifiers are most likely not independent but are correlated. That fact will worsen our results above. In practice, the most improvements in using this method are found when the classifiers used are "very different".

Exercise 2

In-hard voting each classifier in the ensemble provides a single *label* of an instance and ensemble fusion is performed using these labels. A common method is by voting (selecting the class with the largest number of votes). In soft voting each classifier provides a set of numbers which represent what that classifier believes to be the strength of that data sample's membership in the possible target classes. Many times each classifier outputs a set of probabilities. Classifier fusion is done with these set of numbers. Often by averaging or other such numerical procedures.

Exercise 3

This question asks us to think about how hard/easy it would be to parallelize some of the ensemble algorithms from this chapter.

- For bagging, each predictor is built from a random draw (with replacement) of data drawn from the original training set. This sampling can easily be done in parallel.
- For pasting, each predictor is built from a random draw (without replacement) of data drawn from the original training set. In this case we must build our predictor on a smaller dataset that the original one or else we will get the same predictor each time. This can also easily be done in parallel.

- In boosting ensembles, the next tree added to the ensemble is used to predict the error made by the previous ensemble of predictors. Thus one can only fit ensembles like this sequentially.
- In random forests each tree in the ensemble is build from a bootstrap sample of the training dataset. In addition, at each node in the tree the decision the variables we can consider to use for splitting is restricted to a random subset of all possible variables. These procedures are easily parallelized.
- In stacking ensembles, we learn optimal ways to blend the outputs from the base predictors in our ensemble to predict the full outputs of interest. As we must have the output of all of the trained elements in our ensemble it is hard to parallelize this final set. We could parallelize the building of the "first layer" or the individual ensemble predictors but not so easily the learning of their optimal fusion parameters

Exercise 4

The benefit of out-of-bag evaluation is that you are evaluating each trees performance on data that was not used in generating that tree. Thus in using out-of-bad evaluation you get a "free" unbiased estimate of that trees performance on unseen data.

Exercise 5

Extra-trees introduce a random split point at each decision node rather than searching each predictor to find the optimal split point. This gives extra-tree a larger degree of randomness (since the split points values are decided randomly). This extra randomness will mean that each tree will be produced with less influence from the actual dataset and the total ensemble predictor will then have less variance (multiple "samples" of this entire learning algorithm will be quite stable) and more bias (multiple "samples" of this entire learning algorithm will be the same "distance" from the true distribution of the data). This increase in bias and decrease in variance can be good if an original random forest ensemble is overfitting the data. This might happen if we have a situation where we have very few samples m but each sample has a very large number of features n i.e. $m \ll n$.

Extra-trees will be faster to train than random forests as there is no search to find the optimal predictor split point at each node. When making predictions extra-tree and random forest take the same amount of time

Exercise 6

In AdaBoost there is a learning rate parameter named η that controls how much to the jth model affects the final prediction. "Large" values of η will fit the data more tightly with

fewer iterations and potentially overfit the data while values that are too small will cause the algorithm to underfit. If we observe underfitting then we should increase the value of η. Too large a value of η can cause the model to diverge so one should not increase η too much.

One can also increase the number of trees we use to build the ensemble. More trees will allow more complicated patterns to be fit and thus reduce underfitting.

Finally, the degree of overfitting in the ensemble can be controlled by controlling the degree of overfitting of the base predictors. If one reduces the regularization parameters in the base model (reducing the underfitting in the base model) this should help the underfitting problem in the ensemble.

Exercise 7

If you find your gradient boosting predictor is overfitting one can increase the regularization of the base predictors one way (when your base predictors are CART trees) is to reduce the value of the `max_depth` parameter. Another thing to do is to find the optimal number of trees (you might have too many trees if you find you are overfitting). Another thing one can do is to decrease the value of the parameter `learning_rate` as a smaller learning rate parameter avoids overcommitting to the errors in each subsequent fitting.

Chapter 8 (Dimensionality Reduction)

Exercise 1

Reducing a dataset will normally produce a dataset with a smaller number of variables. This means that most algorithms that will be learning off of this reduced dataset will do so faster than the would on the original dataset. The hope is that by reducing dimensions one is mostly removing noise (and not signal). As the noise is not predictive of anything there is no need to "keep it around".

If the dimensionality is reduced to 1-d, 2-d, or 3-d then visualizations of the dataset are possible. This can be a motivation for reducing dimensionality.

Drawbacks could include cases where we have removed signal rather than noise and subsequent learning algorithms would then perform worse. In addition, the transformed features are more difficult to interpret as they are now "blends" of more "base" predictors.

Exercise 2

The curse of dimensionality is the general understanding that increasing the dimension of a system will not necessarily bring algorithmic improvements. Increasing the dimension of the features to a machine learning system is problematic in that now each data sample is very likely to be "very far" from all of the others. Thus as we increase the dimensionality of our dataset we are simultaneously making it more sparse at a much greater rate.

Exercise 3

In general no. If you remove information (even if its noise) it is impossible to "add it back in" as you don't have access to it any longer. Some dimensionality reduction algorithms have a "reverse" method that will give you back a dataset of the same dimension as the original but some information is lost and not all dimensionality reduction algorithms have this method.

Exercise 4

Yes, PCA can be used to reduce the dimension of any dataset. Whether learning using the PCA reduced dataset results in performance that is much worse than learning using the original dataset is problem dependent. The fact that PCA is a linear technique and we are told to assume that our dataset is highly nonlinear indicates that performance on the

reduced dataset will most likely be worse than on the original dataset especially if we are using a nonlinear learning technique that can model the nonlinear relationships.

Exercise 5

If the variance is distributed *uniformly* in the 1000 dimensional space (as it would be if the data was drawn from a 1000 dimensional unit cube) then we would expect to have $1000(0.95) = 950$ remaining dimensions after reduction. Most real world datasets that one would want to learn from have a large degree of "structure" to them so that they are not just uniformly spaced and reducing the dimension to 95% of the variance should result in many fewer final dimensions (than 950).

Exercise 6

Vanilla PCA is a good technique to try if there are no other constrains on the problem and should probability be tried first. Other PCA techniques hope to "solve" some of the problems that you might find with vanilla PCA. For example, incremental PCA is helpful when the full dataset cannot fit into your machines memory. In that case, incremental PCA will feed sequential batches of data (that *will* fit into memory) until the full dataset is processed.

If you think you are only looking for the first d principal components (where $d \ll n$) then randomized PCA will use a randomized algorithm to quickly find these. This can be much faster than using the full SVD algorithm (which computes all of the principle components).

At a heuristic level, kernel PCA uses the "kernel trick" to map the original feature space into an infinite dimensional one which holds nonlinear powers of the features and then applies PCA in that higher dimensional space. This allows kernel PCA to better deal with nonlinearity in the dataset

Exercise 7

If the dimensional reduction technique has an "inverse" method producing points with the same dimension as the original dataset then one can measure the l_1 or the l_2 reconstruction error. Minimizing this error can say to not perform any dimensionality reduction however (as then the reconstruction error is zero). If the dimensionality reduction technique is used as a preprocessing technique then it should keep most of the signal and drop some noise. That means that a learning algorithm trained on the smaller dimensional dataset should perform almost as well as one trained on the full dataset.

Exercise 8

If the dataset has a very large feature dimension n a quick dimensionality reduction algorithm (like using randomized PCA) can produce a much smaller dataset that eliminates many "noise features". A slower more computationally intensive algorithm can then be run on this preprocessed dataset to reduce the dimension further down to only the "informative" features.

Chapter 9 (Unsupervised Learning Techniques)

Exercise 1

Clustering is an algorithmic procedure that aims to group data points into a finite number of "clusters" such that all of the data in a given cluster is "close to" all other data in that same cluster.

Some clustering algorithms are K-means, DBSCAN, agglomerative clustering, BIRCH, Mean-Shift, affinity propagation, spectral clustering, and Gaussian mixtures.

Exercise 2

Clustering can be used for learning more about your data. In this chapter of the book there is some discussion about various uses of clustering techniques. Some from that list include customer segmentation, data analysis, dimensionality reduction, and outlier detection to name a few possibilities.

Exercise 3

One method is the "elbow method" where we plot the inertia of the final model as a function of the number of clusters We then look for a "kink" or an "elbow" in that curve where the addition of another cluster does not cause the inertia to decrease as fast as it had been previously. This is a heuristic technique that one can use to decide on the number of clusters.

Another method is to use the *silhouette scores* for each of several potential clusterings. By plotting the silhouette score as a function of the number of clusters k the cluster number that gives the largest silhouette score is a good number of clusters to use.

Exercise 4

Label propagation is where we have labels for the cluster centers and then use these labels to propagate a label for all samples belonging to the same cluster. To implement this one would cluster our data, label the cluster centers with their corresponding class, and then with the corresponding cluster labels assign labels to all of the data points in each of the clusters.

Exercise 5

Most clustering algorithms have an incremental "batch" version where one can feed sequentially feed the algorithm batches of data (and not the full dataset) to allow it to scale to much larger datasets. K-means has an implementation of this in `sklearn` called `MiniBatchKMeans`.

Clustering algorithms that look for regions of high density include DBSCAN and Gaussian mixture models.

Exercise 6

Active learning is where you ask for help in labeling cases that the current algorithm has a hard time labeling. In detail, the current algorithm is used to predict labels on all unlabeled instances and the ones with the lowest confidence are presented to the user for labeling. Once these instances are labeled we can refit using all labeled instances and repeat by presenting the user the cases which are the most difficult to predict. One could iterate this procedure until the algorithmic improvements are not worth the labeling effort.

Exercise 7

Anomaly detection is used to identify samples that are sufficiently removed from the "bulk" of the data. These samples might be considered outliers and should be inspected/studied to make sure that they are not the cause of some sort of faulty processing or data entry errors.

Operationally novelty detection is done the same way as anomaly detection but the difference is that in novelty detection it is assumed that the model is built using a clean dataset uncontaminated by outliers. In that case samples removed from the "bulk" of the data are "valid" data but could be considered novel samples.

Exercise 8

A Gaussian mixture is a generative data model where the samples of data one observes are assumed to be generated from a number of "sources". Each source is a multidimensional Gaussian random variable. When given the parameters of the generative model one can generate samples. Given data one can estimate the parameters of the mixture. These parameters then provide a soft clustering of the dataset This type of model is good for density estimation, anomaly detection, and soft clustering.

Exercise 9

One can find a model that minimizes an information criterion like BIC or AIC. The `GaussianMixture` class has methods `bic` and `aic` that return these two numbers. Another method one can use is to fit a `BayesianGaussianMixture` with a large number of clusters (more than you expect to actually be found in the data). This later method will set a clusters sampling weight to a very small number if it is not likely to actually *be* a cluster.

Exercise 10

The output of `sklearn.datasets.fetch_olivetti_faces` is a "Bunch" object that has

```
dict_keys(['data', 'images', 'target', 'DESCR'])
```

for its keys.

Next I split the data up into training, validation, and testing datasets. Using the training data I cluster using K-means clustering and look for the optimal number of clusters using the "elbow in the inertia plot". This gives me an "optimal" k of $k = 4$ which is much smaller than the forty different people used in creating the dataset.

I then show the first 10 faces in each of the four clusters using `imshow`. The faces I observe seem to look similar but I think that is somewhat hard to quantify directly. Even in the small view I produced (with only ten faces for each cluster) in each cluster one often sees the same "person" a couple of times. For example, the person with index one had all of their five training faces assigned to the cluster with index three. Many of the forty people had all of their faces assigned to just one cluster. The number of people that had their faces assigned to one, two, and three clusters is given by:

```
How many people were assigned to 1, 2, 3, or 4 clusters:
1    14
2    19
3     7
```

Note that no person had their five faces assigned to more than three clusters.

Exercise 11

To start, I used logistic regression on the image data directly to predict the person in the image. As this dataset is rather small we expect overfitting (which we see below) where I found

```
Training score= 1.000000; Validation score= 0.940299
```

Then as suggested, I first tried to use K-means as a dimensionality reduction tool by using it to map each of the images to distances from some fixed k number of cluster centers. Under these new features the data is of size $m \times k$ and will be of a smaller dimension if $k < n$. I then built a logistic regression model to predict the person in the image using this reduced dataset. For these results I got very poor performance when the number of clusters was quite small. As I increased the value of k results improved

```
k=   31; Training score= 0.9950; Validation score= 0.8358
k=   32; Training score= 0.9950; Validation score= 0.8507
k=   33; Training score= 1.0000; Validation score= 0.8507
k=   34; Training score= 1.0000; Validation score= 0.8657
k=   35; Training score= 1.0000; Validation score= 0.8657
k=   36; Training score= 1.0000; Validation score= 0.8507
k=   37; Training score= 1.0000; Validation score= 0.8507
k=   38; Training score= 1.0000; Validation score= 0.8657
k=   39; Training score= 1.0000; Validation score= 0.8507
k=   40; Training score= 1.0000; Validation score= 0.8657
k=   41; Training score= 1.0000; Validation score= 0.8657
k=   42; Training score= 1.0000; Validation score= 0.8657
k=   43; Training score= 1.0000; Validation score= 0.8806
k=   44; Training score= 1.0000; Validation score= 0.8806
```

Notice that none of these validation scores is as large as the original one indicating that the K-means features by themselves are perhaps not as good of features as the original data.

Next and again following the book, I appended the raw data with the distance to the k cluster centers creating an *expanded* dataset. The results I found were quite surprising. Here for small k the validation score was *better* than any seen thus far. I found

```
k=    1; Training score= 1.0000; Validation score= 0.9701
k=    2; Training score= 1.0000; Validation score= 0.9701
k=    3; Training score= 1.0000; Validation score= 0.9701
k=    4; Training score= 1.0000; Validation score= 0.9701
k=    5; Training score= 1.0000; Validation score= 0.9552
k=    6; Training score= 1.0000; Validation score= 0.9552
k=    7; Training score= 1.0000; Validation score= 0.9552
k=    8; Training score= 1.0000; Validation score= 0.9552
k=    9; Training score= 1.0000; Validation score= 0.9552
```

At this point for larger k the validation results started to get worse. Thus by using one additional feature K-means feature we can increase our accuracy by over 3%.

Exercise 12

I implemented most of this problem in the associated notebook. Fitting the Gaussian mixture model is just an application of using the `sklearn` library. Calling the `sample` method on the resulting Gaussian mixture object gives us an array that we can reshape and view as an image. Some of these images look very realistic but some have "smears" or "scars" on their faces which make them look less like real faces.

The next steps in this exercise are straight forward (but unimplemented). We would transform some of the initial images (or obtain images from an external source). We then would transform the images using the `transform` method on our fitted `pca` object. We could then call the `score_samples` method from our Gaussian mixture model object on these novel images. If our images come from the same set of data that the mixture model was trained on we expect the output from that method to "large" while if the data comes from another distribution we expect the output from `score_samples` to be "small". If we have produced different enough looking images we should be able to detect that they don't look the same as the training images.

Exercise 13

For this exercise I looked at the histogram of the reconstruction error (computed on each image). This is plotted in the notebook but I found statistics on these values to be

```
mse (per image); mean= 0.000187; median= 0.000171; min= 0.000009; max= 0.000421
```

I then took the first training image and rotated it 90 degrees counter-clockwise. This is plotted in the notebook. For this bad image I then transformed it using the PCA `transform` method and then transformed it back to the original image space. This procedure has a reconstruction error of

```
One "bad" image; reconstruction_mse= 0.005352
```

Note that this is more than 10 times the largest reconstruction error seen in the training data. This large magnitude indicates an outlier. If we then plot this reconstructed image we see a face in the normal orientation and not rotated! Given that the starting image was in a different orientation its not surprising that the reconstruction error is so large.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: The XOR operation.

Chapter 10 (Introduction to Artificial Neural Networks with Keras)

Exercise 1

Experimenting with the TensorFlow Playground is a great way to get some *visual* and *intuitive* experience with neural networks. If you've never seen such an application I would highly recommend doing these experiments.

Exercise 2

In my version of the book there was a parenthesis missing from the XOR formula. It should be

$$A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)\,. \tag{6}$$

In *binary* notation the XOR operation can be expressed in Table 1. We can check that the formula given in true in Equation 6 is true in Table 2. Looking at that table we compute each column and then finally the expression given by Equation 6. Note that this column equals the $A \oplus B$ column in Table 6.

Now to evaluate the right-hand-side of Equation 6 graphically in the book we can first compute $C \equiv A \wedge \neg B$ and then $D \equiv \neg A \wedge B$ and then link C and D with a logical or. The book shows how to evaluate C in Figure 10-3. Note that in evaluating this we do it by "summing" and then sending out a "1" if the sum is larger than or equal to 2. Thus if $A = 0$ and $B = 0$ then the formula for C we start by evaluating

$$0 + 0 - 0 = 0\,,$$

which is less than 2 so $C = 0$. If $A = 0$ and $B = 1$ we see that

$$0 + 0 - 1 = -1\,,$$

which is less than 2 so $C = 0$. If $A = 1$ and $B = 0$ we see that

$$1 + 1 - 0 = 2\,,$$

which is greater than or equal to 2 so $C = 1$. Finally, if $A = 1$ and $B = 1$ we see that

$$1 + 1 - 1 = 1\,,$$

which is less than 2 so $C = 0$.

To finish this problem we compute D using the "same" type network as we had for C and then combine C and D using a logical or.

Exercise 3

A perceptron will adjust the parameters of the linear separating hyperplane until the two classes are separated. It will always converge if the points are linearly separable. If the points are not linearly separable then the perceptron (as normally considered) will not converge. In contrast, in logistic regression the problem is convex and there is a well defined minimum that the fitting routine seeks regardless of the separability of the points. Thus in some sense the logistic regression optimum is "well defined". In addition, the logistic regression can be used to estimate class probabilities.

Exercise 4

For some background on notation, a **sigmoidal** function is one that has a characteristic "S" shape. The **logistic** function $l(x)$ given by

$$l(x) = \frac{1}{1 + e^{-x}}\,, \tag{7}$$

is an *example* of such a sigmoidal function The logistic activation function has a particularly simple derivative $l'(x) = l(x)(1 - l(x))$ and thus it was easier to implement the backpropagation algorithm using it since the derivative is always nonzero, continuous, and can be computed in closed form. Having a continuous nonzero derivative means that gradient decent can sequentially step to better and better minimums.

Exercise 5

Three popular activation functions are the logistic function given by Equation 7, the hyperbolic tangent $\tanh(x)$, and the regularized linear unit denoted $\mathrm{ReLU}(x)$.

Exercise 6

This number of neurons in the pass though layer must equal the number of features or $n = 10$. The input matrix would be of size $m \times 10$ where m is the number of samples. The

A	B	$\neg A$	$\neg B$	$A \wedge \neg B$	$\neg A \wedge B$	$(A \wedge \neg B) \vee (\neg A \wedge B)$
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	0

Table 2: The XOR operation from Equation 6.

hidden layers weight vector W_h should be dimensioned such that for the input matrix vector X the vector XW_h should be of dimension $m \times 50$. To this matrix we will add the bias vector/matrix to get $XW_h + B_h$. This means that B_h should be a matrix of size $m \times 50$. We don't want typically want a different vector to be added to each row of XW_h so we would decompose the matrix B_h as

$$B_h = 1_{m\times 1} b_{1\times 50} \,,$$

where $1_{m\times 1}$ is a m dimensional column vector of all ones and $b_{1\times 50}$ is a row vector of length 50. If we can use "broadcasting" in our implementation we might not need to create a matrix B_h but could do something like

$$XW_h + b_{1\times 50} \,.$$

To the above output we will apply our activation function to to get

$$Z \equiv \text{ReLU}(XW_h + b_{1\times 50}) \,,$$

where I have defined the matrix Z above. This matrix is then multiplied on the right by W_o so that

$$ZW_o \,,$$

is a matrix of size $m \times 3$. This means that W_o should be a matrix of size 50×3. To this is added a vector b_o (again using broadcasting) a vector of size 1×3. To the above we would again apply our activation function to get a final output of

$$Y = \text{ReLU}(ZW_o + b_o) \,.$$

This would be a matrix of size $m \times 3$.

Exercise 7

Spam vs. ham is a binary classification problem thus we only need a single output neuron which would represents the probability that the email is spam (the probability an email is ham is then the complement). To obtain this we would use the logistic function as an activation function for this final node.

As MNIST is a classification task where there are ten classes (the digits $0-9$) we would want ten nodes in the output layer. The final activation function in this case would be the softmax function which would convert the linear sums into probabilities for each of the digits.

For regression problems you only need one output neuron without a final activation function as the output does not need to be restricted in anyway.

Exercise 8

Backpropagation refers to the technique by which we train neural networks by estimating the derivative of the networks output nodes with respect to the networks parameters and then use an optimization procedure (like gradient decent) to adjust the network parameters to improve the network's performance on the training data.

Reverse-mode autodiff is an efficient technique used to compute a networks derivative. This computation can then be used as a part of the full backpropagation procedure for model fitting.

Exercise 9

Hyperparameters we can tweak include

- number of hidden layers
- number of neurons in each hidden layer
- learning rate
- batch size
- number of epochs to use in training
- method used to initialize the parameters of the network
- the activation function used at each layer
- regularization method i.e. L_1 or L_2
- regularization coefficient

More layers and more neurons in each layer make the network more likely to overfit. If the network is overfitting reducing these numbers should help. In addition, if the network overfits adding (if its not already there) and increasing the regularization coefficient should reduce overfitting.

Chapter 11 (Training Deep Neural Networks)

Exercise 1

I would think one would have trouble with symmetry breaking if all weights were initiated to the same value regardless of the method used to select that constant.

Exercise 2

Yes. The bias term is a single scalar constant (per layer) that is shared and used to contribute to the sums in each of the neurons in the given layer. The fact that the weights are different (initialized randomly to different constants) means that the bias will affect each neuron slightly differently.

Exercise 3

SELU stands for "scaled ELU" or scaled "exponential linear units". Some of the benefits that SELU have over ReLU are that

- Like ELU in that it has negative values for negative z which helps have the average output closer to zero.
- Like ELU in that it has a nonzero gradient for negative z which helps avoid the dead neuron problem.
- Is "closer" to continuous for $x \approx 0$.
- Under the conditions given in the text, if all of the layers in the network use the SELU activation function the network will "self-normalize" i.e. the output of each layer will have a mean of zero and a standard deviation of one. This will solve the exploding/vanishing gradient problem.

Exercise 4

I would think the different activation function would have benefit in various ways

- SELU activation is a good choice for stacked networks of dense layers (a very common type) and helps to improve the vanishing/exploding gradient problem by self-normalizing the layers.

- leaky ReLU should be used if you need very fast inference (it will be faster than SELU).
- ReLU is a good default activation but is often outperformed by leaky ReLU and SELU.
- tanh and logistic should probably not be used given other more modern (and better choices).
- the logistic activation function could be used as the final layer of a classification problem producing values between zero and one.
- softmax should be used as in the final layer of a multiple output classification problem.

Exercise 5

A momentum value too close to one will give very little change to the $\hat{v}$ vector as new data comes in. This means that momentum values close to one will produce $\hat{v}$ vectors that have low variance but also high bias. Intuitively a momentum value close to one means "fully propagate the known information" and "ignore" new information.

This can result in the SGD algorithm overshooting the minimum and ending up "cross" the bowl shaped error surface. Further iterations will cause the estimates to oscillate around the minimum location. This can cause the total optimization procedure to take more time than it would have otherwise undoing all of these oscillations.

Exercise 6

A sparse model can be produced by using

- Force small weights after training to zero. Note that this is an unprincipled technique and will most likely result in a worse model on the training dataset (and perhaps the testing set).
- Strong l_1 regularization during training.
- The TensorFlow Model Optimization Toolkit (TF-MOT) has an API that is able to produce sparse models.

Exercise 7

Dropout can slow training significantly but the final model can be much better than one trained without dropout. During inference there is no slowdown as the final weights are used (none are zeroed). In MC dropout we set **training=True** when making predictions and do this B number of times. This means that on each prediction we will randomly drop

some number of neurons (set their outputs equal to zero) and get a prediction under that configuration. We will then average the predictions from all of these B networks. This will increase inference making it B times slower.

Exercise 8-10 (book & notebook)

The exercises at the end of this chapter only had one computational problem (deep learning on the CIFAR10 dataset) while the Jupyter notebooks that I originally downloaded with the book had three:

- Exercise 8: Building DNNs on the MNIST dataset
- Exercise 9: Transfer learning
- Exercise 10: Pretraining on an auxiliary task

In the notebooks that came with the book these problems were unworked. Upon further study these were found to be the problems from the *first* edition of this book. There the version of TensorFlow used was 1.x while in the second edition of this book it is TensorFlow 2.x. As it seemed like these would be valid problems to work in TensorFlow 2.x I've worked them here and their solutions can be found in the ipython notebook for this chapter.

Exercise 8: Building DNNs on the MNIST dataset:

To start, I build a DNN on all of the digits as specified: using five hidden layers of 100 neurons each, He initialization, and the ELU activation function. After fitting this model gave

```
5 layers 100 neurons (digits 0-9): validation accuracy= 0.940600; testing accuracy= 0.937500
```

Next, I split the data into the first five digits and build a model to predict their labels (I changed the optimization procedure to Adam optimization with early stopping but kept the underlying network the same). This gave

```
5 layers 100 neurons (digits 0-4): validation accuracy= 0.992572; testing accuracy= 0.993968
```

Note that this result is significantly better than the previous result.

Next I tried to find optimal hyperparameters for this DNN by keeping the Adam optimizer but changing the hyper-parameters that determine how the DNN is built. Specifically I changed the number of hidden layers, the number of neurons in each hidden layer, and three parameters specific to the Adam optimizer (the learning rate and the parameters β_1 and β_2). Even though one can search over values for β_1 and β_2 I don't think its very

productive. It seems that for many choices for these parameters (i.e. when they are different from the defaults) the resulting DNN gives very poor performance. Searching over the learning rate to use in Adams optimization does seem to provide some benefit however. Using `RandomizedSearchCV` to look for optimal parameters we finally end with

```
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 55}
```

which gives an final accuracy of

```
validation accuracy (after hyperparameter optimization) = 0.992572; testing accuracy= 0.992995
```

This is again an improvement.

Next we apply batch normalization to the "default" DNN by placing `BatchNormalization` layers after each `Dense` layer. Fitting this model gives

```
validation accuracy (after batch normalization)= 0.987490; testing accuracy= 0.989881
```

which seems slightly worse than the result we obtained after hyperparameter optimization. We could perhaps get better results using batch normalization if we also attempted to optimize our hyperparameters. Plots of the learning curves indicate that learning is quite fast in this case (converges after only a few epochs).

Next we add dropout. From what I can tell this can be done with or without batch normalization. The results under either way seem very similar

```
validation accuracy (with Dropout only) = 0.993354; testing accuracy= 0.992216
validation accuracy (with Dropout+BN only) = 0.988272; testing accuracy= 0.990854
```

From these results it seems that dropout by itself (without batch normalization) seems to give better results.

Exercise 9: Transfer learning:

Here I load one of the models trained above (five hidden layers followed by batch normalization), "clone" it, add a new final softmax layer, and turn off training for all but the final (softmax) layer. I then trained over 10 epochs trying to recognize the digits 5-9. The validation accuracy started poor but ended up with an accuracy of about 0.68.

I was not sure what the definition of caching but I found that if I unfroze the initial *five* dense layers and trained again for another 10 epochs, the accuracy after this method was around 0.921. We expect to achieve a much better accuracy as we now have many more weights to learn and we are not really using "transfer learning" as much in this case. As

these results were *so* much better than the previous attempt (using transfer learning with a small modification) I think what I did was not the intended result.

Instead, it seemed more relevant to the idea of "transfer learning" to make more small scale changes to the initial model and then add only few trainable layers "on-top-of" this initial model. In order to do this I then copied the first *four* layers (with batch normalization) while dropping the last hidden layer and then again adding a new softmax output node. I then froze all but the last softmax layer and fit for 10 epochs. This gave an accuracy of around 0.76 on the validation data. Note that this is a slight improvement in accuracy over just appending a softmax layer (where we obtained a validation accuracy of 0.68). This also adds a bit of an argument to the idea that in a deep neural network the initial layers are often feature extractor layers and the higher layers combined these features to perform the final discrimination.

I then unfroze the top *two* layers and refit for another 10 epochs. The total accuracy on the validation data was then found to be 0.87. This is significantly larger than the previous two results and was obtained using a very small training set. Thus we have been able to improve our accuracy while keeping much of the existing network unchanged. This seems to be more inline with the idea of "transfer learning".

Exercise 10: Pretraining on an auxiliary task:

Here I followed the suggested graph building instructions. Because the routine I wrote to generate training data produced datasets with a large memory footprint I implemented a loop where I would generate training and validation data, train the model for some number of epochs with early stopping, and then repeat this procedure for several times in a loop each time generating new sets of training/validation data. This seemed to be a valid way of handling datasets that are larger than be held entirely in main memory. Once I finished training I was able to get an accuracy of around 0.81 on the validation set and 0.89 on the testing set.

Chapter 12 (Custom Models and Training with TensorFlow)

Exercise 1

TensorFlow is an open source numerical library aimed at doing computations with *tensors* or multidimensional arrays of numbers. It has been coded to run on CPU's, GPU's, and special purpose hardware. It has strong support for large-scale machine learning applications specifically building, training, and running neural networks.

Other popular deep learning libraries include PyTorch and Theano and others.

Exercise 2

TensorFlow is not a drop in replacement for numpy. TensorFlow can be specified to run on various hardware (for improved performance) while numpy runs primarily on the CPU. TensorFlow does offer many of the functionality that numpy offers but the design decisions in implementing TensorFlow were made to make it more suited for the large-scale numerical computations needed for machine learning.

There are also a few differences between the two API's. The names of methods that do the same operation can be different (i.e. `tf.reduce_sum` vs. `np.sum`) and the results of various operations can be somewhat different (i.e. `transpose` in TensorFlow vs. `T` in numpy).

Exercise 3

The default number of bits for variables in TensorFlow is 32 while in numpy the default number of bits is 64. Thus the first case our integers are 32-bit while in the second they are 64-bit.

Exercise 4

Some other data structures in TensorFlow (besides `Tensor`) include

- `tf.SparseTensor`
- `tf.TensorArray`
- `tf.RaggedTensor`

- `tf.string`
- `tf.sets`
- `tf.queue`

Exercise 5

Writing a function is a very simple way to implement a loss function. When saving your model you will have to manually remember any parameters that went into the function(such as thresholds) and the function definition must be available when loading the model.

Writing a class that subclasses from the `keras.losses.Loss` class allows more uniform behavior in the saving and loading of models. By implementing the `get_config` method to return any needed parameters subsequent call made to the `from_config` method (when the model is loaded) will load the correct parameters.

Exercise 6

If the mean of the desired metric over an epoch can be computed as the mean of the metric over the batches then a function argument to the `metrics` call will work. If the desired metric is a streaming metric (stateful metric) then you need to subclass the `keras.metrics.Metric` class.

Exercise 7

For any custom layer one might define one could create a custom model that implemented the same functionality. The choice of one or the other depends on how simple it is to understand the code written. The layer class is for defining components of your model and the model class is for defining how the layers stack together.

Exercise 8

Some architectures require more control over the fitting procedure and would require one to write your own custom fitting routines (these would often be research models). In addition, writing a custom fitting routine can create a better understand how the default `fit` method works.

Exercise 9

It is preferable to have the components convertible to TF Functions. There are methods one can use to include arbitrary python code but they need to be treated somewhat specially.

Exercise 10

From the section in this chapter on `TF Function Rules` one common theme is to "be careful calling external libraries".

Exercise 11

A dynamic Keras model might be helpful if one is debugging as one can use a python debugger to debug ones code. Making a model dynamic, however, will prevent TensorFlow from using any of its graph features which will slow down the model training and inference.

Exercise 12

Part (a): See the ipython notebook where we implement a custom layer as requested.

Part (b-c): As this layer is very similar (exactly the same) as a `keras.layers.LayerNormalization` layer I show that numerically that the output from the layer I implement and the Keras layer gives the same results. Next I build a simple NN without my layer (to get a baseline for performance) and then a NN that uses my layer. Training the first net gives

```
Initial model: test loss(MSE)= 0.441, test mae= 0.473
```

Training the second net (the one with my custom layer) gives

```
MyNormalizingLayer model: test loss(MSE)= 0.412, test mae= 0.468
```

As the results are very similar we have an indication that adding layer normalization will not help this problem much and that these two models give similar results on this dataset.

Exercise 13

Part (a): To implement this we can write code similar to that given in the book. There we implement a custom training loop and compare the loss and accuracy obtained with using

the more standard Keras `fit` method.

On the test set the standard Keras method gives

```
ReLU model: test loss= 0.514, test accuracy= 0.824
```

while the custom fitting code gives

```
Custom fit model: test loss= 0.437, test accuracy= 0.858
```

As the results on the test set are very similar we will take that as an indication that our custom fitting routine was implemented correctly.

Chapter 13 (Loading and Preprocessing Data with TensorFlow)

Exercise 1

Loading and preprocessing data is an important part of building machine learning models. The `tf.data` API makes many of the common operations easier to perform and does not require one to write processing code for each project.

Exercise 2

Several files would be easier to transport and process in parallel in that most processing steps are easier to perform when working with several smaller files rather than one huge one.

Exercise 3

If loading a batch of data is much slower than training on that batch then the pipeline would be the bottleneck. TensorBoard has some visualization tools that can help. If you find that your GPU is not working at 100% then your input processing is likely a bottleneck. Reworking the input processing to run in parallel threads can help speed this up.

Exercise 4

One can save any binary data to a TFRecord file as a TFRecord file is just a sequence of binary records. Each records can/could be a *protobuf* and there are various reasons why that would be advantageous.

Exercise 5

Converting ones data to the `Example` protobuf format (rather than a custom protobuf format) will allow you to use the `tf.io.parse_single_example` (and other) methods to parse protobufs in that format. If it is not sufficient general for your application one can define ones own format but there will be more work to be done.

Exercise 6

Compression helps most if the data will be read over a network connection where bandwidth is limited. Doing it systematically will slow down some processing as the CPU will need to spend time compressing and uncompressing. Compression helps if the transmit time of the raw data is larger than the time required to uncompress the compressed data.

Exercise 7

The more preprocessing that is done "in the model" the slower training will be as that preprocessing has to be done on each batch. If one can preprocess the data and remove any needed computation from the model building part the entire training will be faster. Of course any preprocessing done in this way will also need to be done to all new data and separating the preprocessing and model evaluation runs the risk of introducing errors if the preprocessing is not done to the raw data before being presented to the model. If the preprocessing is done "in the model" then there is less risk of an error of this type.

Exercise 8

One can encode them "as is" that is using an integer (numeric) representation for the feature value. This would be the desired approach if *differences* in the values of two features represented something of meaningful. If the integers represented color then this would not be true. If they represented intensity of some process then it would be.

Another way to encode these is using one-hot encoding. This would be the valid choice if the integers represented something that could not easily be compared like color in the above example.

Another way to encode these would be with an embedding layer. In that case each unique integer is mapped to a vector in $\mathbb{R}^e$ for some embedding dimension e and that mapping is learned during the model fitting.

Text can be encoded in several different ways. A "bag-of-words" approach is one method, another is by using term frequency-inverse document frequency (tf-idf) encoding, and yet another is via embedding (currently the state of the art). For word/sentence encodings using embedding the dimension e can be quite large and is learned during model fitting.

Chapter 14 (Deep Computer Vision Using Convolutional Neural Networks)

Exercise 1

Modern image processing uses CNN for many reasons

- A CNN can take advantage of the spatial structure in an image where a DNN cannot (without additional architecture design).
- A CNN can use many fewer parameters than a full connected DNN because the feature maps have parameters that can be shared when the feature map is applied to different regions of the image. This makes it less likely to overfit and faster to train.
- Feature maps in a CNN can "recognize" the same feature in different parts of the image making it easier for the CNN to generalize.

Exercise 2

In the first layer, each 3×3 kernel move though a single image which has a depth of three (for the RGB components). This means it has $3 \times 3 \times 3 = 27$ weights. To this we add one bias weights for a total of 28 weights/parameters. The lowest layer then will have $100 \times 28 = 2800$ weights.

A single input to the middle layer is an "image" from the lowest layer which has a depth of 100 and so a 3×3 kernel will have $3 \times 3 \times 100 = 900$ weights plus a bias term for a total of 901 weights. The middle layer will then have $901 \times 200 = 180200$ weights.

A single input to the top layer is an "image" from the middle layer will has a depth of 200 and so a 3×3 kernel will have $3 \times 3 \times 200 = 1800$ weights plus a bias term for a total of 1801 weights. The top layer will then have $1801 \times 400 = 720400$ weights.

This is a total of

$$2800 + 180200 + 720400 = 903400\,.$$

The weights will take 903400×32 bits or

$$\frac{903400 \times 32}{8} = 3613600\,,$$

bytes or

$$\frac{3613600}{10^6} = 3.6136\,,$$

megabytes.

As we have a stride of two the "dimensions" of the feature maps will be reduced by two at each step. As the original image is 200×300 the size of the feature maps at each layer is

$$100 \times 150\,, \quad 50 \times 75\,, \quad 25 \times 38\,,$$

where $38 \approx \frac{75}{2}$. Given the number of feature maps held at each layer the number of floats held is

$$100 \times 100 \times 150\,, \quad 200 \times 50 \times 75\,, \quad 400 \times 25 \times 38\,.$$

This is 2630000 floats. As each occupies 32 bits or 4 bytes this is $\frac{2630000(4)}{10^6} = 10.52$ megabytes. To this we need to add the memory storage requirements of the weights computed above This gives around $10.52 + 3.6 = 14.12$ megabytes.

Note that the above is an overestimate as once we have computed a layer we can dispose of it and can release the memory it occupied.

On a mini-batch of 50 we would multiply the above by 50.

Exercise 3

If our GPU runs out of memory then we can

- Make the mini-batchs size smaller.
- Make the kernel sizes smaller (fewer parameters to learn).
- Add some max pooling layers to introduce down-sampling.
- Reduce dimensionality by increasing the stride.
- Remove some horizontal layers.
- Use variables with smaller word sizes (i.e. single vs double precision).
- Distribute the CNN computation across multiple devices.

Exercise 4

As the two layers reduce the feature maps by the same amount there is no memory advantage of one over the other.

A max pooling layer can provide some level of translation invariance over the image as the maximum over a region does not change as the kernel is moved slightly around the pixel that gives the maximum output. Another reason to prefer a max pooling layer is that it has no parameters. This means that introducing one does not increase the number of parameters we have to learn.

Exercise 5

Local feature normalization was a transformation first applied to the feature maps in AlexNet to allow the most strongly signaling neuron at one depth of the output feature maps to *inhibit* the activation of the neurons at the same row and column as the strongest. This forces each layer to learn different features forcing the strong output to be localized.

Exercise 6

The book discusses the various changes as each new network was considered.

LeNet-5 used convolutions and average pooling nets to process the image followed by a fully connected softmax layer. I believe it was the first to consider using convolutional processing parts to process images.

AlexNet stacked convolutional layers on top of each other (rather than following each convolutiona1 layer with a pooling layer) and several fully connected layers at the top to provide lower level feature interpretation. AlexNet used dropout on some of the upper layers and data augmentation to provide a more diverse training dataset. Finally AlexNet introduced a "competitive normalization" step immediately after the ReLU step the goal of which was to localize strong responses in a given feature map (and dampen them in adjacent features maps).

GoogLeNet was *much* deeper than AlexNet with the introduction of inception modules which allow the sharing of parameters. The network is somewhat complex and is described in better detail in the text.

ResNet was a very deep network that had "skip connections" that allowed the input vector x to be combined with several layers of processing. These skip connections enable all layers of the network to start learning right away (as their inputs are far from zero on average).

Xception introduced the ideas of using convolutional filters that are separable (i.e. one dimensional). These separable convolutional layers use fewer parameters and less memory than nonseparable layers.

SENet which stands for Squeeze-and-Excitation Network. This architecture essentially extends inception and ResNet architectures. The network is somewhat complicated and is described in more detail in the text.

Exercise 7

In a fully convolutional network we replace the dense topmost layer with another convolutional layer.

Exercise 8

In semantic segmentation we seek to classify each and every pixel in an image. The main difficulty in using CNN for semantic segmentation is that the image looses resolution as it moves up the network flow. This is especially true with pooling layers and convolutional layers with larger strides (greater than one). The detailed pixel information needs to be retained somehow for use in the higher level classification performed at the topmost layers.

Chapter 15 (Processing Sequences Using RNNs and CNNs)

Exercise 1

WWX: Working here.

Chapter 16 (Natural Language Processing with RNNs and Attention)

Exercise 1

WWX: Working here.

Chapter 17 (Autoencoders, GANs, and Diffusion Models)

Exercise 1

WWX: Working here.

Chapter 18 (Reinforcement Learning)

Exercise 1

WWX: Working here.