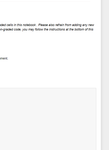
Practice Lab: Advice for Applying Machine Learning

In this lab, you will explore techniques to evaluate and improve your machine learning models.

Outline

- 1 - Packages
- 2 - Evaluating a Learning Algorithm (Polynomial Regression)
  - 2.1 Splitting your data set
  - 2.2 Error calculation for model evaluation, linear regression
    - Exercise 1
  - 2.3 Compare performance on training and test data
- 3 - Bias and Variance
  - 3.1 Plot Train, Cross-Validation, Test
  - 3.2 Finding the optimal degree
  - 3.3 Tuning Regularization
  - 3.4 Getting more data: Increasing Training Set Size (m)
  - 3.5 Evaluating a Learning Algorithm (Neural Network)
  - 4.1 Data Set
  - 4.2 Regularized cost function
    - Exercise 2
  - 4.3 Regularized gradient
  - 4.4 Learning and accuracy evaluation
  - 4.5 Tips

NOTE: To preserve work from the autograder, you are not allowed to edit or delete non-graded cells in this notebook . Please also refrain from adding any new cells. Once you have passed this assignment and want to experiment with any of the non-graded code, you may follow the instructions at the bottom of this notebook.



1 - Packages

First, let's run the cell below to import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing.
- matplotlib is a plotting library compatible with numpy.
- sklearn is a basic library for data science.
- scipy.io is a package platform for machine learning.

```
In [1]: import numpy as np
        %matplotlib inline
        from matplotlib import matplotlib pyplot
        from sklearn.linear_model import LinearRegression, Ridge
        from sklearn.preprocessing import PolynomialFeature, StandardScaler
        from sklearn.model_selection import train_test_split
        from sklearn.metrics import mean_squared_error
        from scipy.io import loadmat
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Dense
        from tensorflow.keras.activations import relu,linear
        from tensorflow.keras.losses import SparseCategoricalCrossentropy
        from tensorflow.keras.optimizers import Adam

        import logging
        logging.getlogger("tensorflow").setLevel(logging.ERROR)

        from public_tests_a1 import *
        from lab_utils_common import dlblue, dlorange
        from sklearn.preprocessing import PolynomialFeatures
        from sklearn.pipeline import Pipeline

        tf.autograph.set_verbosity(0)
```

2 - Evaluating a Learning Algorithm (Polynomial Regression)

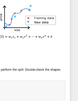
Let's say you have created a machine learning model and you find it fits your training data very well.

Congratulations! But wait. The goal of creating the model was to be able to predict values for new examples.

How can you see your model's performance on new data before deploying it?

The answer has been:

- Split your original data set into "Training" and "Test" sets.
- Fit the model to the training data, minimizing $J_{train}(\vec{w},b)$
- Use the test data to evaluate the model on new data.
- Develop an error function to evaluate your model.



$$J_{test}(\vec{w},b) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(f_{\vec{w},b}(\vec{x}_{test}^{(i)}) - y_{test}^{(i)})^2$$

2.1 Splitting your data set

Lecture version training: 67%% of our data set for training. Let's use an sklearn function train_test_split to perform the split. Double-check the shapes after running the following cell.

```
In [2]: # generate some data
        m,n,degrees,noise = gen_data(m=10, n=1)
        print(f"shape of data: {m} {n} {degrees} {noise}")

        split_frac = 0.67
        X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=1-split_frac,)
        print(f"shape of X_train: {X_train.shape}, y_train.shape: {y_train.shape}")
        print(f"shape of X_test.shape: {X_test.shape}, y_test.shape: {y_test.shape}")

        m_train =len(y_train [0])
        m_test =len(y_test [0])
        print(f"number of training data: {m_train} {m_test}")
```

2.1.1 Plot Train, Test sets

You can see below the data points that will be part of training (in red) are intermixed with those that the model is not trained on (blue). This particular data set is a quadratic function with some added. The "Noise" curve is shown for reference.

```
In [3]: fig, ax = plt.subplots(1,1,figsize=(4,4))
        ax.scatter(X_train, y_train, marker='x', c='red', label="y_train")
        ax.set_title("Training, Test",fontsize=14)
        ax.set_xlabel("x")
        ax.set_ylabel("y")

        ax.scatter(X_test, y_test, marker = 'o', c='blue', label="y_test")
        ax.scatter(X_train, y_train, color = dlblue,'Noise', label="Noise")
        ax.legend(loc='upper left')
        plt.show()
```

Figure 1

2.2 Error calculation for model evaluation, linear regression

Mean squaring a linear regression model, you average the squared error difference of the predicted values and the target values.

$$J_{cv}(\vec{w},b) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(f_{\vec{w},b}(\vec{x}_{cv}^{(i)}) - y_{cv}^{(i)})^2 \tag{1}$$

Exercise 1

Below, create a function to evaluate the error on a data set for a linear regression model.

```
In [5]: # UNQ C1
        # GRADED FUNCTION: eval_mse
        def eval_mse(y, yhat):
            """
            Calculate the mean squared error on a data set.
            Args:
              y (ndarray): Shape (m,) or (m,1) Target value of each example
              yhat (ndarray): Shape (m,) or (m,1) prediction value of each example
            Returns:
              err (scalar):
            """
            m = len(y)
            err = 0.0
            for i in range(m):
                yhat_i = yhat[i]
                y_i = y[i]
                err = err + (yhat_i - y_i)**2
            err = err / (2*m)
            return(err)

        ans_mse = eval_mse(y, y)
        print(ans_mse)

        # UNIT TESTS
        # eval_mse(y, y, 2)

        # DO NOT EDIT
        # Click the button to reset your notebook
```

Click the button to reset your notebook.

2.3 Compare performance on training and test data

Let's build a high degree polynomial model to minimize training error. This will use the linear_regression functions from sklearn. The code is for the important utility that if you would like to see the details. The steps below are:

- create and fit the model. (fit is another name for training or running gradient descent).
- compute the error on the training data.
- compute the error on the test data.

```
In [6]: # create a model, in sklearn, fitted on training data
        degree = 10
        lmodel = lin_model(degree)
        lmodel.fit(X_train, y_train)

        # predict on training data, find training error
        yhat = lmodel.predict(X_train)
        err_train = lmodel.meas(y_train, yhat)

        # predict on test data, find error
        yhat = lmodel.predict(X_test)
        err_test = lmodel.meas(y_test, yhat)
```

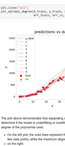
The computed error on the training set is substantially less than that of the test set.

```
In [8]: print("Training set the error,MSE,for polynomial regression model of degree", lmodel.d,": ",
                err_train,"Test set the error,MSE,for polynomial regression model of degree", lmodel.d,": ",
                err_test)
```

```
Training set the error,MSE,for polynomial regression model of degree 10 :  0.0 Test set the error,MSE,for polynomial regression model of degree 10 :  inf
```

The following plot shows why this is. The model fits the training data very well. To do so, it has created a complex function. The test data was not part of the training and the model does a poor job of predicting on this.
This model would be described as: Overfitting. (It has high variance) Or generalized poorly.

```
In [9]: # plot predictions over data range
        x = np.linspace(0,int(X_train[-1]),100)[:,reshape(-1,1)
        y_pred = lmodel.predict(x).reshape(-1,)

        plt_train_test(X_train, y_train, X_test, y_test, x, y_pred, x_ideal, y_ideal, degree)
```



The test set error shows this model will not work well on new data. If you use the test error to guide improvements to the model, then the model will perform well on the test data, but you will have no data to tell how it will perform on another set of data it has never data performance.

The proposed model training technique is to separate data into three groups. The distribution of training, cross-validation and test sets shown in the below table is a typical distribution, but can be varied depending on the amount of data available.

	data	% of data	Description
training	$(\vec{x}_{train}, \vec{y}_{train})$	60%	Data used to tune model parameters w and b in training or fitting
cross validation	$(\vec{x}_{cv}, \vec{y}_{cv})$	20%	Data used to tune other model parameters like degree of polynomial, regularization or the architecture of a neural network
test	$(\vec{x}_{test}, \vec{y}_{test})$	20%	Data used to test the model after tuning to gauge performance on new data

Let's generate these data sets below. We'll once again use train_test_split from sklearn but will call it twice to get three splits

```
In [10]: # generate data
         m,n,degrees,noise = gen_data(m=5, n=1)
         print(f"shape of data: {m} {n} {degrees} {noise}")

         split_frac1 = 0.60
         X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=1-split_frac1,)
         print(f"shape of X_train: {X_train.shape}, y_train.shape: {y_train.shape}")
         print(f"shape of X_test.shape: {X_test.shape}, y_test.shape: {y_test.shape}")

         split_frac2 = 0.50
         X_cv, X_test, y_cv, y_test = train_test_split(X_test,y_test,test_size=split_frac2,)
         print(f"shape of X_cv: {X_cv.shape}, y_cv.shape: {y_cv.shape}")
         print(f"shape of X_test.shape: {X_test.shape}, y_test.shape: {y_test.shape}")

         m_train =len(y_train [0])
         m_cv =len(y_cv [0])
         m_test =len(y_test [0])
         print(f"number of training data: {m_train} {m_cv} {m_test}")
```

3 - Bias and Variance

Above, it was shown the impact of the polynomial model on training and test error. Here, we look at the impact of the degree of polynomial on the training and cross-validation performance. We'll use an example.

3.1 Plot Train, Cross-Validation, Test

You can see below the datapoints that will be part of training (in red) are intermixed with those that the model is not trained on (blue) and (c).

```
In [11]: fig, ax = plt.subplots(1,1,figsize=(4,4))
         ax.scatter(X_train, y_train, marker='x', c='red', label="y_train", 3c=)
         ax.set_title("Training, CV, Test",fontsize= 14)
         ax.set_xlabel("x")
         ax.set_ylabel("y")

         ax.scatter(X_train, y_train, color = 'red', label="y_train")
         ax.scatter(X_cv, y_cv, marker = 'o', c='blue', label="y_cv")
         ax.scatter(X_test, y_test, color = dlblue,'Noise', label="Noise")
         ax.legend(loc='upper left')
         plt.show()
```

Figure 1

3.2 Finding the optimal degree

In previous labs, you found that you could create a model capable of fitting complex curves by utilizing a polynomial (See Course1, Week2 Feature Engineering and Polynomial Regression Lab). Further, you demonstrated that by increasing the degree of the polynomial, you could create overfitting. (See Course1, Week2 Cost Function Lab). Let's see that increasing the fit is not our ability to find the difference between overfitting and underfitting.

Let's loop through repeatedly, increasing the degree of the polynomial each iteration. Here, we're going to use the sklearn linear regression model for speed and simplicity.

```
In [12]: max_degree = 9
         err_train = np.zeros(max_degree)
         err_cv = np.zeros(max_degree)
         x = np.linspace(0,int(X_train[-1]),100)
         y_ideal = np.sin(x)*np.pi/2 + np.abs(x)*np.pi/5

         for degree in range(max_degree):
             lmodel = lin_model(degree)
             lmodel.fit(X_train, y_train)
             yhat = lmodel.predict(X_train)
             err_train[degree] = lmodel.meas(y_train, yhat)
             yhat = lmodel.predict(X_cv)
             err_cv[degree] = lmodel.meas(y_cv, yhat)
             x = np.linspace(0,int(X_train[-1]),100)
             y_pred = lmodel.predict(x)

         optimal_deg_idx = np.argmin(err_cv)

         plt_optimal_fit(X_train, y_train, X_cv, y_cv, x, y_ideal, err_train, err_cv, optimal_deg_idx, lambda_range)
```

The plot above demonstrates that separating data into two groups, data the model is trained on and data the model has not been trained on, can be used to determine the degree of overfitting or underfitting. In our example, we created a variety of models varying from underfitting to overfitting by increasing the degree of the polynomial regression.

- On the left plot, the solid lines represent the predictions from these models. A polynomial model with degree 1 produces a straight line that intersects very few data points, while the maximum degree levels very closely to every data point.
- on the right,
  - the dotted lines are the trained cross-validation error, as the model complexity increases as expected
  - the end of the cross-validation data decreases initially as the model starts to conform to the data, but then increases as the model starts to overfit on the training data (bias to generalize).

It's worth noting that the curves in these examples are not smooth as one might draw for a lecture. It's clear the specific data points assigned to each group can change your results significantly. The general trend is what is important.

3.3 Tuning Regularization.

In previous labs, you found sklearn regularization to reduce overfitting. Similar to degree, one can use the same methodology to tune the regularization parameter, lambda.

Let's demonstrate this by starting with a high degree polynomial and varying the regularization parameter.

```
In [14]: lambda_range = np.array([0,1e-4,1e-3,1e-2,1e-1,1,10,1e2,1e3])
         m = len(lambda_range)
         degree = 10
         err_train = np.zeros(m)
         err_cv = np.zeros(m)
         x = np.linspace(0,int(X_train[-1]),100)
         y_ideal = np.sin(x)*np.pi/2 + np.abs(x)*np.pi/5

         for i in range(m):
             lambda_= lambda_range[i]
             lmodel = lin_model(degree, regularization=True, lambda_=lambda_)
             lmodel.fit(X_train, y_train)
             yhat = lmodel.predict(X_train)
             err_train[i] = lmodel.meas(y_train, yhat)
             yhat = lmodel.predict(X_cv)
             err_cv[i] = lmodel.meas(y_cv, yhat)
             y_pred[i,:] = lmodel.predict(x)

         optimal_reg_idx = np.argmin(err_cv)

         plt_tune_regularization(X_train, y_train, X_cv, y_cv, x, y_ideal, err_train, err_cv, optimal_reg_idx, lambda_range)
```


Above, the plots show that as regularization increases, the model moves from a high variance (overfitting) model to a high bias (underfitting) model. The vertical line in the right plot shows the optimal value of lambda. In this example, the polynomial degree was set to 10.

3.4 Getting more data: Increasing Training Set Size (m)

When a model is overfitting (high variance), collecting additional data can improve performance. Let's try that here.

```
In [16]: X_train, X_cv, X_test, y_train, y_cv, y_test = gen_data(m=500, n=1, noise=10)
         print(f"shape of X_train: {X_train.shape}, y_train.shape: {y_train.shape}")
```

The above plots show that when a model has high variance and is overfitting, adding more examples improves performance. Note the curves on the left plot. The thick curve with the highest error is for a smooth curve that fits to the center of the data. On the right, as the number of examples increases, the performance of the training set and cross-validation set converge to similar values. Note that the curves are not as smooth as one might see in a lecture. That is to be expected. The more realistic curves have more data to tune generalization.

Note that adding more examples when the model has high bias (underfitting) does not improve performance.

4 - Evaluating a Learning Algorithm (Neural Network)

Above, you tuned aspects of a polynomial regression model. Here, you will work with a neural network model. Let's start by creating a classification data set.

4.1 Data Set

We'll use this data set to generate a data set and split it into training, cross-validation (CV) and test sets. In this example, we're increasing the percentage of cross-validation data points for emphasis.

```
In [17]: # generate and split data set
         X,y,centers,std = gen_blobs()

         # split the data. Large CV population for demonstration
         split_frac = 0.6
         X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=1-split_frac,)
         split_frac = 0.1/(1.0-split_frac)
         X_cv, X_test, y_cv, y_test = train_test_split(X_test,y_test,test_size=split_frac,)

         print(f"shape of X_train: {X_train.shape}, y_train.shape: {y_train.shape}")

         X_train_stack, X_cv_stack, X_test_stack = stack_features(X_train, X_cv, X_test)
```


Above, you can see the data on the left. There are six clusters identified by color. Both training points (dots) and cross-validation points (triangles) are shown. The ideal performance would be to correctly identify the cluster membership of the cross-validation points (triangles). What would you expect a neural network model to do? Which would be an example of overfitting? underfitting?
On the right is an example of an "ideal" model, or a model that might create bounding the source of the data. The lines represent "equal distance" boundaries where the distance between points and the center is equal. It's worth noting that this model would "misassign" roughly 8% of the total points.

4.2 Evaluating categorical model by calculating classification error

The evaluation function for categorical models used here is simply the fraction of incorrect predictions.

$$J_{cv} = \frac{1}{m_{cv}}\sum_{i=1}^{m_{cv}} \begin{cases} 1 & if\ \hat{y}^{(i)} \neq y^{(i)} \\ 0 & otherwise \end{cases}$$

Exercise 2

Below, complete the routine to calculate classification error. Note, in this data, target values are the index of the category and are not one-hot encoded.

```
In [20]: # UNQ C2
         # GRADED FUNCTION: eval_cat_err
         def eval_cat_err(y, yhat):
             """
             Calculate the categorization error
             Args:
               y (ndarray): Shape (m,) or (m,1) Target value of each example
               yhat (ndarray): Shape (m,) or (m,1) prediction value of each example
             Returns:
               cerr (scalar):
             """
             m = len(y)
             incorrect = 0
             for i in range(m):
                 yhat_i = yhat[i]
                 y_i = y[i]
                 if yhat_i != y_i:
                     incorrect = incorrect + 1
             cerr = incorrect / m
             return(cerr)

         ans_cerr = eval_cat_err(y, y)
         print(ans_cerr)

         # UNIT TESTS
         # eval_cat_err(y, y, 2)

         # DO NOT EDIT
         # Click the button to reset your notebook
```

Click the button to reset your notebook.

```
In [21]: # generate data set
         m,n,centers,std = gen_blobs(n=2, seed=42)

         # split the data. Large CV population for demonstration
         split_frac = 0.6
         X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=1-split_frac,)
         split_frac = 0.1/(1.0-split_frac)
         X_cv, X_test, y_cv, y_test = train_test_split(X_test,y_test,test_size=split_frac,)

         print(f"shape of X_train: {X_train.shape}, y_train.shape: {y_train.shape}")

         X_train_stack, X_cv_stack, X_test_stack = stack_features(X_train, X_cv, X_test)
```


Above, you can see the data on the left. There are six clusters identified by color. Both training points (dots) and cross-validation points (triangles) are shown. The ideal performance would be to correctly identify the cluster membership of the cross-validation points (triangles). What would you expect a neural network model to do? Which would be an example of overfitting? underfitting?
On the right is an example of an "ideal" model, or a model that might create bounding the source of the data. The lines represent "equal distance" boundaries where the distance between points and the center is equal. It's worth noting that this model would "misassign" roughly 8% of the total points.

4.2 Evaluating categorical model by calculating classification error

The evaluation function for categorical models used here is simply the fraction of incorrect predictions.

$$J_{cv} = \frac{1}{m_{cv}}\sum_{i=1}^{m_{cv}} \begin{cases} 1 & if\ \hat{y}^{(i)} \neq y^{(i)} \\ 0 & otherwise \end{cases}$$

Exercise 2

Below, complete the routine to calculate classification error. Note, in this data, target values are the index of the category and are not one-hot encoded.

```
In [20]: # UNQ C2
         # GRADED FUNCTION: eval_cat_err
         def eval_cat_err(y, yhat):
             """
             Calculate the categorization error
             Args:
               y (ndarray): Shape (m,) or (m,1) Target value of each example
               yhat (ndarray): Shape (m,) or (m,1) prediction value of each example
             Returns:
               cerr (scalar):
             """
             m = len(y)
             incorrect = 0
             for i in range(m):
                 yhat_i = yhat[i]
                 y_i = y[i]
                 if yhat_i != y_i:
                     incorrect = incorrect + 1
             cerr = incorrect / m
             return(cerr)

         ans_cerr = eval_cat_err(y, y)
         print(ans_cerr)

         # UNIT TESTS
         # eval_cat_err(y, y, 2)

         # DO NOT EDIT
         # Click the button to reset your notebook
```

Click the button to reset your notebook.

```
In [21]: # generate data set
         m,n,centers,std = gen_blobs(n=2, seed=42)

         # split the data. Large CV population for demonstration
         split_frac = 0.6
         X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=1-split_frac,)
         split_frac = 0.1/(1.0-split_frac)
         X_cv, X_test, y_cv, y_test = train_test_split(X_test,y_test,test_size=split_frac,)

         print(f"shape of X_train: {X_train.shape}, y_train.shape: {y_train.shape}")

         X_train_stack, X_cv_stack, X_test_stack = stack_features(X_train, X_cv, X_test)
```

```
y_hat = np.array([1, 1, 1])
y = np.array([0.99, 0.99, 0.99])
# EXPECTED OUTPUT
# CAT MET FOOD
categorization error 0.111, expected:0.111
categorization error 0.250, expected:0.250
    All tests passed.
```

Click for hints

5 - Model Complexity

Below, you will build two models. A complex model and a simple model. You will evaluate the models to determine if they are likely to overfit or underfit.

5.1 Complex model

Exercise 3

Below, compose a three-layer model:

- Dense layer with 120 units, relu activation
- Dense layer with 40 units, relu activation
- Dense layer with 6 units and a linear activation (not softmax)
- Compile using
- loss with SparseCategoricalCrossentropy, remember to use from_logits=True
- Adam optimizer with learning rate of 0.01.

```
In [20]: # UNQ_C3
         # GRADED CELL: model
         import tensorflow as tf
         import logging
         logging.getLogger("tensorflow").setLevel(logging.ERROR)

         tf.random.set_seed(1234)
         model = Sequential(
             ### START CODE HERE ###
             Dense(120, activation = 'relu'),
             Dense(40, activation = 'relu'),
             Dense(6, activation = 'linear')
             ### END CODE HERE ###
         , name="Complex"
         )
         model.compile(
         ### START CODE HERE ###
         loss= tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True),
         optimizer= tf.keras.optimizers.Adam(learning_rate = 0.01),
         ### END CODE HERE ###
         )

In [22]: # BEGIN UNIT TEST
         model.fit(
             X_train, y_train,
             epochs=1000
         )
         # END UNIT TEST
```

```
Epoch 1/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.8252
Epoch 992/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0153
Epoch 993/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0142
Epoch 994/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0109
Epoch 995/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0231
Epoch 996/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0109
Epoch 997/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0109
Epoch 998/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0168
Epoch 999/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0315
Epoch 1000/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.0152
```

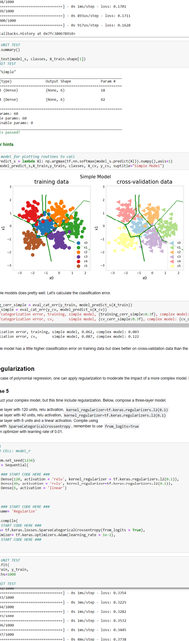
```
Out[22]: <keras.callbacks.History at 0x7fc1e5bf1d0>
```

```
In [23]: # BEGIN UNIT TEST
         model.summary()
         model_test(model, classes, X_train.shape[1])
         # END UNIT TEST
```

```
Model: "Complex"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 120)               360

 dense_2 (Dense)             (None, 40)                4840

 dense_3 (Dense)             (None, 6)                 246

=================================================================
Total params: 5,446
Trainable params: 5,446
Non-trainable params: 0
_________________________________________________________________
    All tests passed!
```

Click for hints

```
In [24]: # make a model for plotting routines to call
         model_predict = lambda X1: np.argmax(tf.nn.softmax(model.predict(X1).numpy(),axis=1)
         plt_nn(model,X_train,y_train, classes, X_tr, y_tr, suptitle="Complex Model")
```



This model has worked very hard to capture outliers of each category. As a result, it has misclassified some of the cross-validation data. Let's calculate the classification error.

```
In [25]: training_cerr_complex = eval_cat_err(y_train, model_predict(X_train))
         cv_cerr_complex = eval_cat_err(y_cv, model_predict(X_cv))
         print("training categorization error, {:5f}, complex model: {:5f}".format(training_cerr_complex,tr_nf))
         print("categorization error, {:5f}, complex model: {:5f}".format(cv_cerr_complex,cv_nf))
```

```
categorization error, training, complex model: 0.083
categorization error, cv,       complex model: 0.122
```

5.1 Simple model

Now, let's try a simple model

Exercise 4

Below, compose a two-layer model:

- Dense layer with 6 units, relu activation
- Dense layer with 6 units and a linear activation. Compile using
- loss with SparseCategoricalCrossentropy, remember to use from_logits=True
- Adam optimizer with learning rate of 0.01.

```
In [26]: # UNQ_C4
         # GRADED CELL: model_s

         tf.random.set_seed(1234)
         model_s = Sequential(
             ### START CODE HERE ###
             Dense(6, activation = 'relu'),
             Dense(6, activation = 'linear')
             ### END CODE HERE ###
         , name= "Simple"
         )
         model_s.compile(
         ### START CODE HERE ###
         loss= tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True),
         optimizer= tf.keras.optimizers.Adam(learning_rate = 0.01),
         ### END CODE HERE ###
         )

In [27]: import logging
         logging.getLogger("tensorflow").setLevel(logging.ERROR)

         # BEGIN UNIT TEST
         model_s.fit(
             X_train_s, y_train,
             epochs=1000
         )
         # END UNIT TEST
```

```
Epoch 1/1000
1/1] [==============================] - 0s 844us/step - loss: 0.3165
Epoch 991/1000
1/1] [==============================] - 0s 845us/step - loss: 0.1672
Epoch 992/1000
1/1] [==============================] - 0s 844us/step - loss: 0.1560
Epoch 993/1000
1/1] [==============================] - 0s 834us/step - loss: 0.1395
Epoch 994/1000
1/1] [==============================] - 0s 885us/step - loss: 0.1678
Epoch 995/1000
1/1] [==============================] - 0s 966us/step - loss: 0.1680
Epoch 996/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.1791
Epoch 997/1000
1/1] [==============================] - 0s 855us/step - loss: 0.1711
Epoch 998/1000
1/1] [==============================] - 0s 917us/step - loss: 0.1628
```

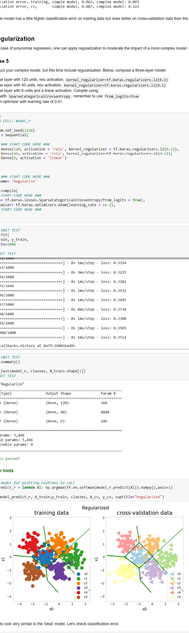
```
Out[27]: <keras.callbacks.History at 0x7fc1b8e78010>
```

```
In [28]: # BEGIN UNIT TEST
         model_s.summary()
         model_test(model_s, classes, X_train.shape[1])
         # END UNIT TEST
```

```
Model: "Simple"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 6)                 18

 dense_6 (Dense)             (None, 6)                 42

=================================================================
Total params: 60
Trainable params: 60
Non-trainable params: 0
_________________________________________________________________
    All tests passed!
```

Click for hints

```
In [29]: # make a model for plotting routines to call
         model_predict_s = lambda X1: np.argmax(tf.nn.softmax(model_s.predict(X1).numpy(),axis=1)
         plt_nn(model_predict_s,X_train,y_train, classes, X_tr, y_tr, suptitle="Simple Model")
```



This simple models does pretty well. Let's calculate the classification error.

```
In [30]: training_cerr_simple = eval_cat_err(y_train, model_predict_s(X_train))
         cv_cerr_simple = eval_cat_err(y_cv, model_predict_s(X_cv))
         print("training categorization error, {:5f}, simple model: {:5f}".format(training_cerr_simple,tr_nf)
         print("categorization error, {:5f},       simple model: {:5f}".format(cv_cerr_simple,cv_nf))
```

```
categorization error, training, simple model: 0.083, complex model: 0.083
categorization error, cv,       simple model: 0.067, complex model: 0.122
```

Our simple model has a little higher classification error on training data but does better on cross-validation data than the more complex model.

6 - Regularization

As in the case of polynomial regression, one can apply regularization to moderate the impact of a more complex model. Let's try this below.

Exercise 5

Reconstruct your complex model, but this time include regularization. Below, compose a three-layer model:

- Dense layer with 120 units, relu activation, kernel_regularizer=tf.keras.regularizers.l2(0.1)
- Dense layer with 40 units, relu activation, kernel_regularizer=tf.keras.regularizers.l2(0.1)
- Dense layer with 6 units and a linear activation. Compile using
- loss with SparseCategoricalCrossentropy, remember to use from_logits=True
- Adam optimizer with learning rate of 0.01.

```
In [31]: # UNQ_C5
         # GRADED CELL: model_r

         tf.random.set_seed(1234)
         model_r = Sequential(
             ### START CODE HERE ###
             Dense(120, activation = 'relu', kernel_regularizer = tf.keras.regularizers.l2(0.1)),
             Dense(40, activation = 'relu', kernel_regularizer=tf.keras.regularizers.l2(0.1)),
             Dense(6, activation = 'linear')
             ### END CODE HERE ###
         , name= "Regularized"
         )
         model_r.compile(
         ### START CODE HERE ###
         loss= tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True),
         optimizer= tf.keras.optimizers.Adam(0.01),
         ### END CODE HERE ###
         )

In [32]: # BEGIN UNIT TEST
         model_r.fit(
             X_train, y_train,
             epochs=1000
         )
         # END UNIT TEST
```

```
Epoch 1/1000
1/1] [==============================] - 0s 1ms/step - loss: 8.3354
Epoch 991/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.3225
Epoch 992/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5262
Epoch 993/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.3512
Epoch 994/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.3405
Epoch 995/1000
1/1] [==============================] - 0s 4ms/step - loss: 0.3758
Epoch 996/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.3388
Epoch 997/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.3585
Epoch 1000/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.3514
```

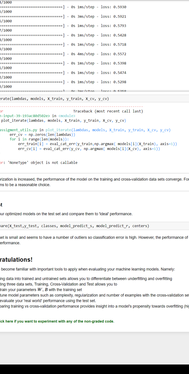
```
Out[32]: <keras.callbacks.History at 0x7fc1b8ed8ad0>
```

```
In [33]: # BEGIN UNIT TEST
         model_r.summary()

         model_r_test(model_r, classes, X_train.shape[1])
         # END UNIT TEST
```

```
Model: "Regularized"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_9 (Dense)             (None, 120)               360

 dense_8 (Dense)             (None, 40)                4840

 dense_7 (Dense)             (None, 6)                 246

=================================================================
Total params: 5,446
Trainable params: 5,446
Non-trainable params: 0
_________________________________________________________________
    All tests passed!
```

Click for hints

```
In [34]: # make a model for plotting routines to call
         model_predict_r = lambda X1: np.argmax(tf.nn.softmax(model_r.predict(X1).numpy(),axis=1)

         plt_nn(model_predict_r,X_train,y_train, classes, X_tr, y_tr, suptitle="Regularized")
```



The results look very similar to the 'ideal' model. Let's check classification error.

```
In [35]: training_cerr_reg = eval_cat_err(y_train, model_predict_r(X_train))
         cv_cerr_reg = eval_cat_err(y_cv, model_predict_r(X_cv))
         test_cerr_reg = eval_cat_err(y_test, model_predict_r(X_test))
         print("training categorization error, {:5f}, simple model: {:5f}, complex model: {:5f}".format(training_cerr_reg,tr_nf), simple model: {training_cerr_simple:6.4f}, complex model: {tra
         print("categorization error, {:5f},       regularized: {:5f}".format(cv_cerr_reg,cv_nf), simple model: {cv_cerr_simple:6.4f}, complex model: {cv_
```

```
categorization error, training, regularized: 0.072, simple model, 0.083, complex model: 0.083
categorization error, cv,       regularized: 0.061, simple model, 0.067, complex model: 0.122
```

The simple model is a bit better in the training set than the regularized model but worse in the cross-validation set.

7 - Iterate to find optimal regularization value

As you did in these regressions, you can try many regularization values. This code takes several minutes to run. If you have time, you can run it and check the results. If not, you have completed the graded parts of the assignment.

```
In [36]: tf.random.set_seed(1234)
         lambda_vec = np.array([0.0, 0.01, 0.1, 0.2, 0.3])
         models=[None] * len(lambda_vec)

         for i in range(len(lambda_vec)):
             lambda_ = lambda_[i]
             models[i] = Sequential(
                 ...
                 Dense(120, activation = 'relu', kernel_regularizer=tf.keras.regularizers.l2(lambda_)),
                 Dense(40, activation = 'relu', kernel_regularizer=tf.keras.regularizers.l2(lambda_)),
                 Dense(6,activation = 'linear')
             )
             models[i].compile(
             loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
             optimizer=tf.keras.optimizers.Adam(0.01),
             )

             models[i].fit(
             X_train, y_train,
             epochs=1000
             )
             print("Finished lambda = {lambda_}")
```

```
Epoch 991/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5938
Epoch 992/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5921
Epoch 993/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5793
Epoch 994/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5428
Epoch 995/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5758
Epoch 996/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5572
Epoch 997/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5398
Epoch 998/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5474
Epoch 999/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5208
Epoch 1000/1000
1/1] [==============================] - 0s 1ms/step - loss: 0.5398
Finished lambda = 0.0
Finished lambda = 0.01
Finished lambda = 0.1
Finished lambda = 0.2
Finished lambda = 0.3
```

```
In [37]: plot_iterate(lambda_, models, X_train, y_train, X_cv, y_cv)
```

```
Type\nTrue             True\nCat\nnot cat\ncat (all last)
cv\n1-J(w)\n1-J(w)\nlast\n(all last)
<<< plot_iterate(lambda_, models, X_train, y_train, X_cv, y_cv)

/usr/local/lib/python3.6/dist-packages/matplotlib/axes/_base.py:3183: UserWarning: Ignoring fixed x limits to fulfill fixed data aspect with adjustable data spl
  warnings.warn("Ignoring fixed %s limits to fulfill fixed data aspect with adjustable data "
```