Applied Deep Learning - Part 2: Real World Case Studies

Arden Dertat Sep 1, 2017 · 17 min read

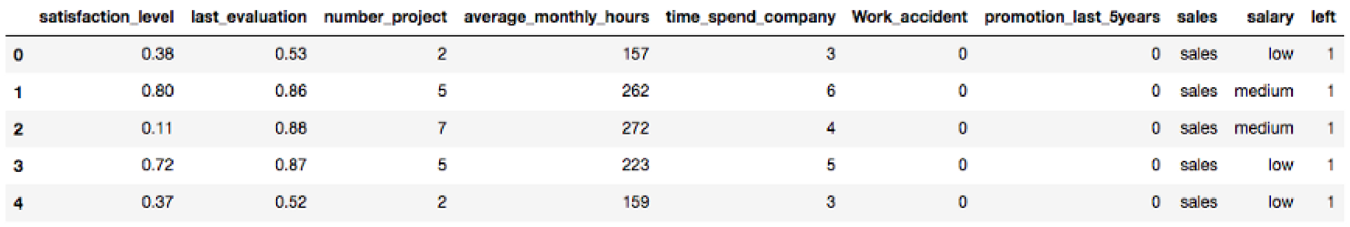
Overview

Welcome to Part 2 of Applied Deep Learning series. Part 1 was a hands-on introduction to Artificial Neural Networks, covering both the theory and application with a lot of code examples and visualization. Now comes the cool part, end-to-end application of deep learning to real-world datasets. We will cover the 3 most commonly encountered problems as case studies: binary classification, multiclass classification and regression.

The code for this article is available here as a Jupyter notebook, feel free to download and try it out yourself.

1. Case Study: Binary Classification

We will be using the Human Resources Analytics dataset on Kaggle. We're trying to predict whether an employee will leave based on various features such as number of projects they worked on, time spent at the company, last performance review, salary etc. The dataset has around 15,000 rows and 9 columns. The column we're trying to predict is called "left". It's a binary column with 0/1 values. The label 1 means that the employee has left.



	satisfaction_level	last_evaluation	number_project	average_monthly_hours	time_spend_company	Work_accident	promotion_last_5years	sales	salary	left
0	0.38	0.53	2	157	3	0	0	sales	low	1
1	0.80	0.86	5	262	6	0	0	sales	medium	1
2	0.11	0.88	7	272	4	0	0	sales	medium	1
3	0.72	0.87	5	223	5	0	0	sales	low	1
4	0.37	0.52	2	159	3	0	0	sales	low	1

1.1) Data Visualization & Preprocessing

First things first, let's perform some data visualization and preprocessing before jumping straight into building the model. This part is crucial, since we need to know what type of features we are dealing with. For every ML task, we at least need to answer the following questions:

- What type of features do we have: real valued, categorical, or both?
- Do any of the features need normalization?
- Do we have null values?

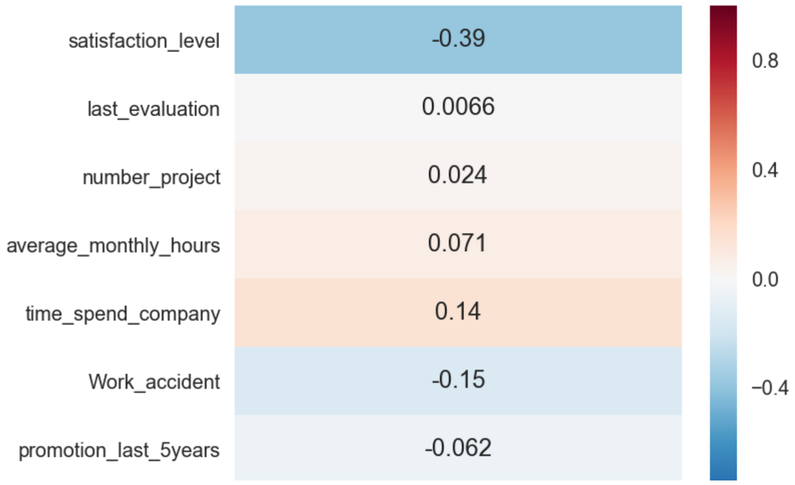
- What is the label distribution, are the classes imbalanced?
- Is there a correlation between the features?

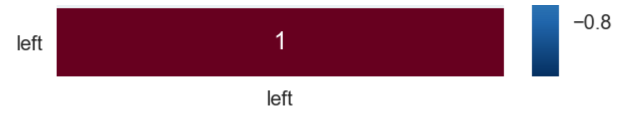
The jupyter notebook contains the detailed analysis. In summary, there are both real and categorical features. There are no null values, but some features need normalization. 76% percent of the examples are labeled as 0, meaning the employee didn't leave.

Let's check the correlation of the features with the labels (the column named "left"). We will use the *seaborn* package for the correlation plot.

```
seaborn.heatmap(rawdf.corr()[['left']], annot=True, vmin=-1, vmax=1)
```

case_classification_1.py hosted with ♡ by GitHub view raw



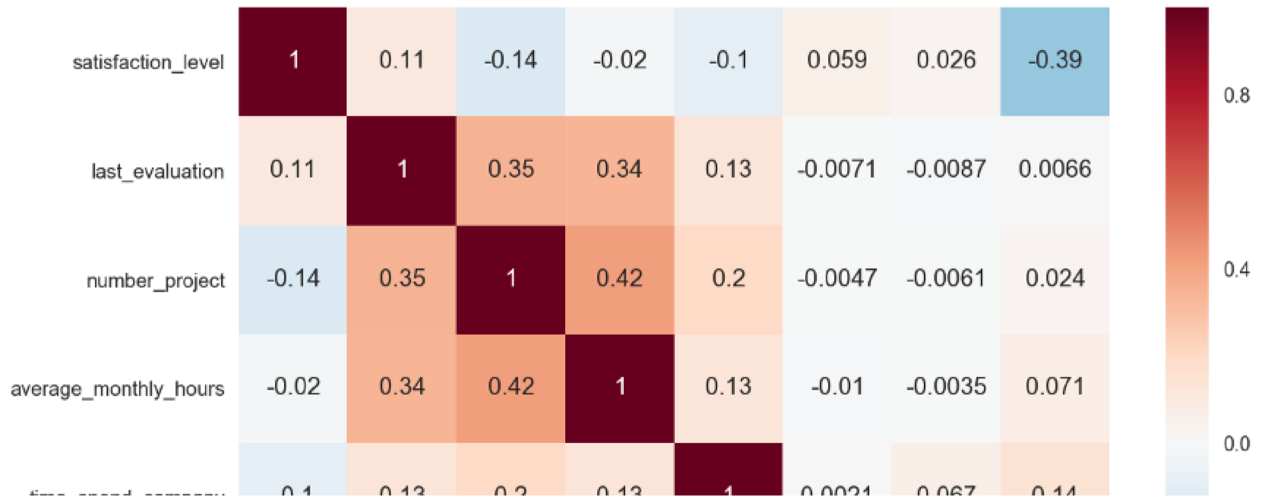


In this plot, positive values represent correlation and negative values represent inverse correlation with the label. Of course "left" has perfect correlation with itself, you can ignore that. Other than that only one feature has a strong signal, which is the "satisfaction_level", inversely correlated with whether the employee has left. Which makes sense.

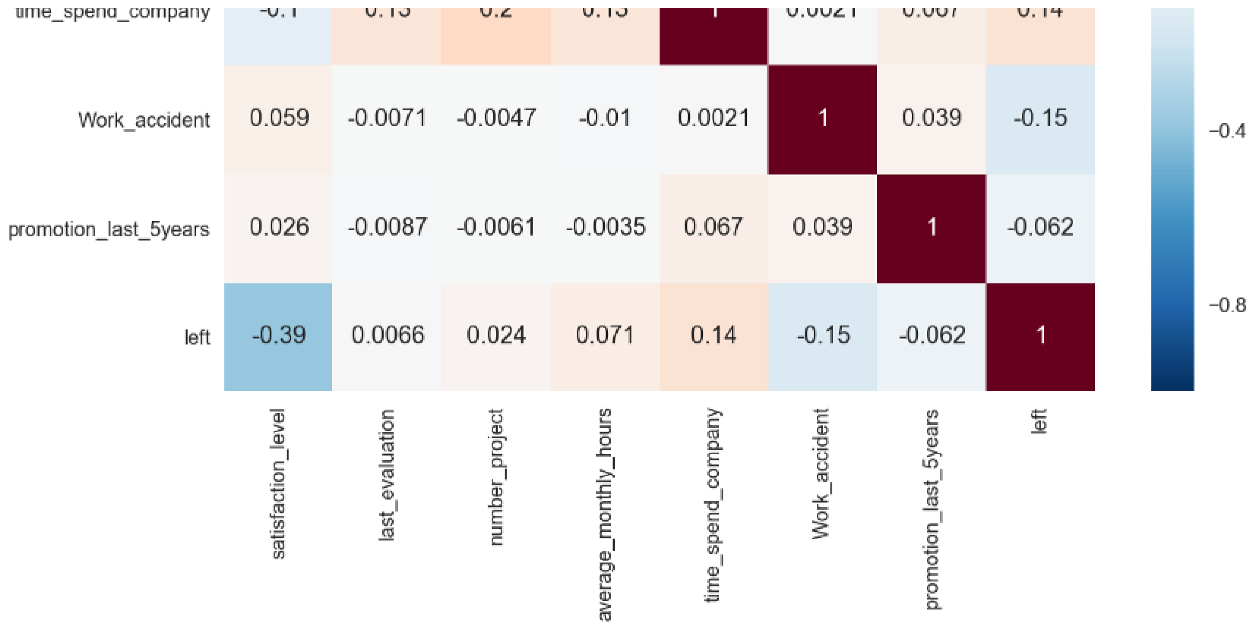
Now let's look at the pairwise correlation of all features with one another.

```python
seaborn.heatmap(rawdf.corr(), annot=True, square=True, vmin=-1, vmax=1)
```

case_classification_2.py hosted with ♡ by GitHub view raw



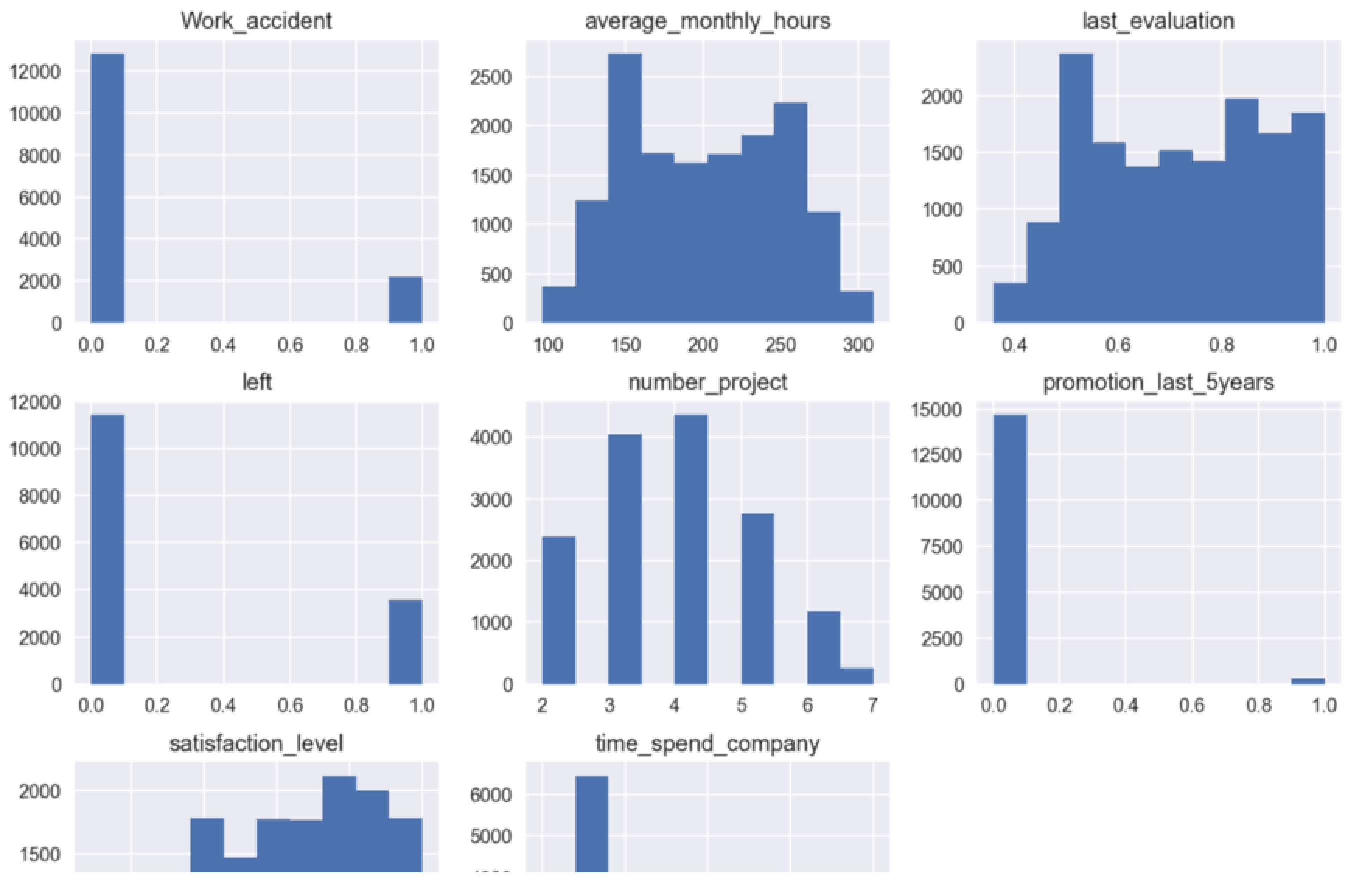
satisfaction_level	1	0.11	-0.14	-0.02	-0.1	0.059	0.026	-0.39
last_evaluation	0.11	1	0.35	0.34	0.13	-0.0071	-0.0087	0.0066
number_project	-0.14	0.35	1	0.42	0.2	-0.0047	-0.0061	0.024
average_monthly_hours	-0.02	0.34	0.42	1	0.13	-0.01	-0.0035	0.071

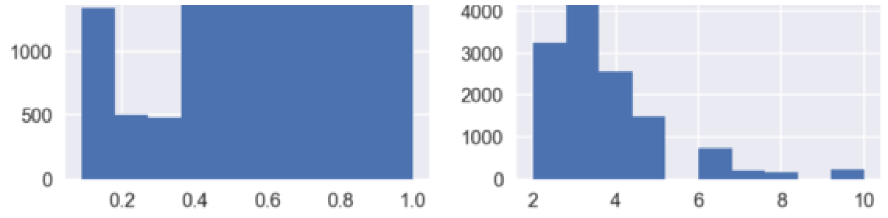


We see that "average_monthly_hours" is positively correlated with "number_project", which again makes sense. The more projects a person is involved with, the more hours of work they need to put in.

Now let's look at the distribution of feature values. By inspecting the histograms of features we can see which ones need normalization. What's the motivation behind this? What does normalization mean and why is it needed? Most ML algorithms perform better if the real valued features are scaled to be in a predefined range, for example [0, 1]. This is particularly important for deep neural networks. If the input features consist of large values, deep nets really struggle to learn. The reason is that as the data flows through the layers, with all the multiplications and additions, it gets large very quickly and this negatively affects the optimization process by saturating non-linearities. We will see the detailed demonstration of this in another article, for now we need to pay attention to feature values to be small numbers.

Looking at feature histograms, we need to normalize 3 of the features: average_monthly_hours, number_project, and time_spend_company. All other features are within [0, 1] so we can leave them alone.





Scikit-learn has several normalization methods, what we will use is *StandardScaler*. It individually scales the features such that they have zero mean and unit variance, so they all belong to a standard *Normal(0, 1)* distribution. Note that this doesn't change the ordering of the feature values, it just changes the scale. It's a simple yet extremely important trick.

The data we loaded is in a *pandas DataFrame*. Pandas is an extremely popular package to deal with tabular data, especially in an interactive environment like jupyter notebooks. DataFrame is the most commonly used data structure of pandas that acts as a container for our data, and exposes several built-in functions to make our life easier (check out the notebook for more details). In the code snippet below, df is the DataFrame for our data.

```python
ss = StandardScaler()
scale_features = ['average_monthly_hours', 'number_project', 'time_spend_company']
df[scale_features] = ss.fit_transform(df[scale_features])
```

case_classification_3.py hosted with ♡ by GitHub view raw

Scikit-learn API is very well designed and contains 4 very commonly used methods. Predictors are ML models like Logistic Regression, and transformers are data manipulators like Standard Scaler.

- *fit*: For predictors performs training on the given input. For transformers computes the statistics like mean and standard deviation of the input to be used later.
- *transform*: For transformers manipulates the input data using the stats learned by the fit function. We run the transform method after fit since there's a dependency. Predictors don't support this method.
- *fit_transform*: Performs fit + transform in a single call efficiently. For transformers, computes the stats of the input and performs the transformation. It's very a commonly used method with transformers. For predictors, trains the model and performs prediction on the given input.
- *predict:* As its name suggests, for predictors performs the prediction task using the model trained with the fit method. Very commonly used with predictors. Transformers don't support this method.

Now that we have scaled the real-valued features to be in a desirable range, let's deal with the categorical features. We need to convert categorical data to *one-hot* representation. For example the salary column contains 3 unique string values: low, medium and high. After one-hot conversion we will have 3 new binary columns: salary_low, salary_medium and salary_high. For a given example, only one of them will have the value 1, the others will be 0. We will then drop the original salary column because we don't need it anymore.

The one-hot conversion is performed by the *get_dummies* of pandas. We could have also used the *OneHotEncoder* in scikit-learn, they both get the job done. Since our data is already is in a pandas dataframe, get_dummies is easier. It also automatically perform the renaming of the features.

```python
categorical_features = ['sales', 'salary']
df_cat = pd.get_dummies(df[categorical_features])
df = df.drop(categorical_features, axis=1)
df = pd.concat([df, df_cat], axis=1)
```

case_classification_4.py hosted with ♡ by GitHub view raw

salary_high	salary_low	salary_medium
0	1	0
0	1	0
0	0	1
0	0	1
1	0	0

Now comes the final part of creating the training and test data. The model will perform learning on the training set and be evaluated on the held-out test set. Scikit-learn has a convenient *train_test_split* function. We only need to specify the fraction of the test set, in our case 30%. But first we convert our data from pandas dataframe to numpy array using the *values* attribute of the dataframe.

```python
X = df.drop('left', axis=1).values
y = df['left'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

case_classification_5.py hosted with ♡ by GitHub view raw

1.2) Logistic Regression Model

Now that we're done with the data preprocessing and train/test set generation, here comes the fun part, training the model. We first start with a simple model, Logistic Regression (LR). We will then train a deep ANN and compare the results to LR.

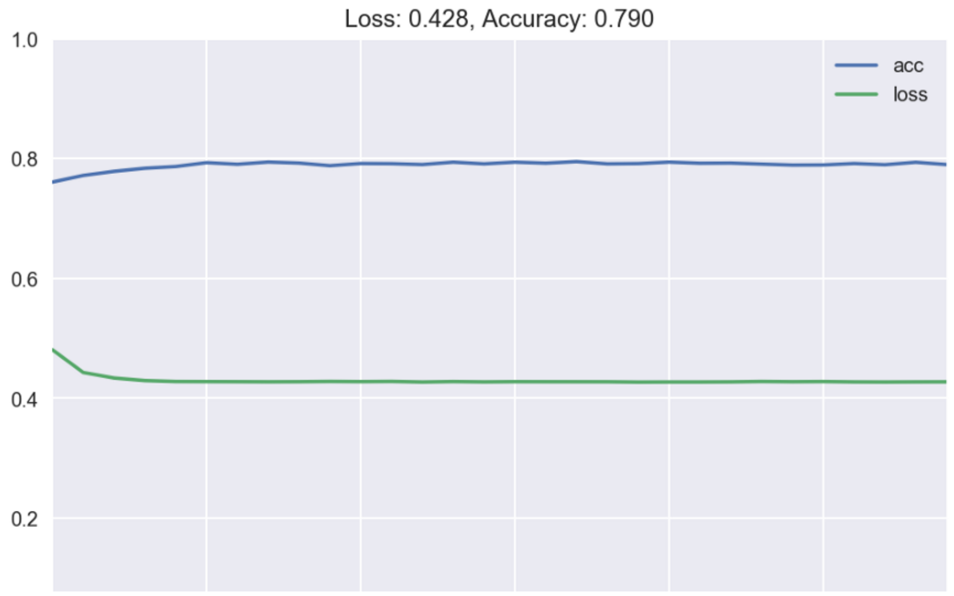
After the first article, building the model should be very familiar.

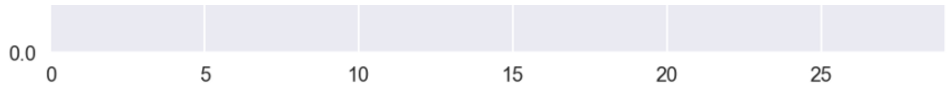
```python
lr_model = Sequential()
```

```
lr_model.add(Dense(1, input_shape=(X_train.shape[1],), activation='sigmoid'))

lr_model.compile(Adam(lr=0.01), 'binary_crossentropy', metrics=['accuracy'])

lr_history = lr_model.fit(X_train, y_train, verbose=1, epochs=30)
plot_loss_accuracy(lr_history)
```

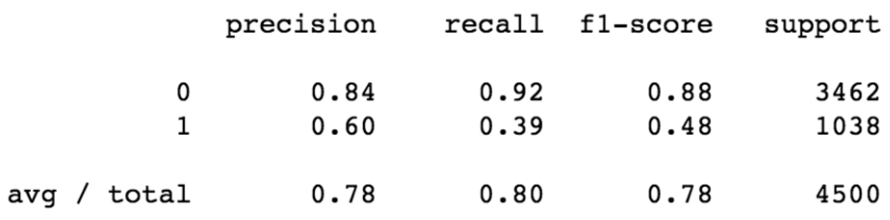
case_classification_6.py hosted with ♡ by GitHub view raw

We get 79% training accuracy. This is actually pretty bad, because above we saw that 76% of the labels were 0. So the most naive classifier which always outputs 0 regardless of the input would get 76% accuracy, and we're not doing much better than that. This means our data is not linearly separable, just like the examples we saw in the first article, and we need a more complex model.

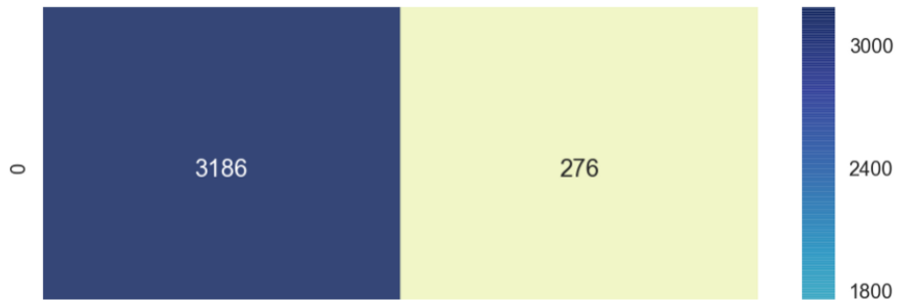


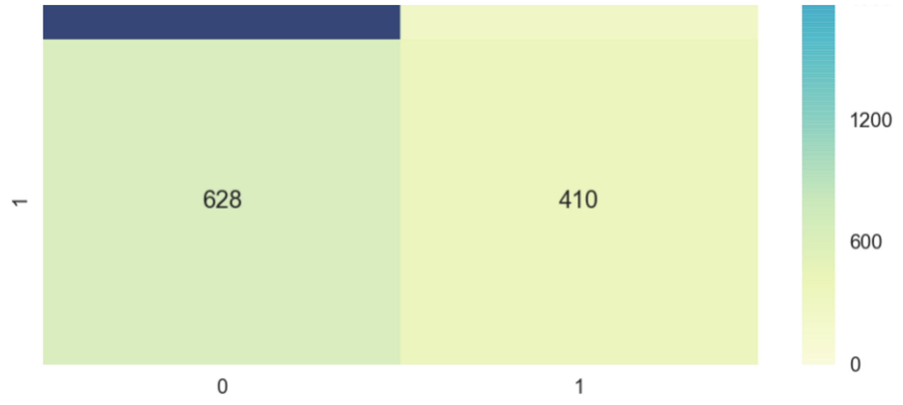


Above chart depicts the training loss and accuracy. But more importantly we're interested in the metrics of the test set. Metrics in the training set might be misleading since the model is already trained on it, we want to check how the model performs on an held-out test set. Test accuracy is 78%, slightly lower than training accuracy. Test accuracy of ML models are almost always less than training, because the test data is unseen to the model during the training process. Looking at the classification report, we see that only 60% of the examples belonging to class 1 are classified correctly. Pretty bad performance. The confusion matrix also doesn't look promising showing a lot of misclassified examples.



```
             precision    recall  f1-score   support

          0       0.84      0.92      0.88      3462
          1       0.60      0.39      0.48      1038

avg / total       0.78      0.80      0.78      4500
```





1.3) ANN Model

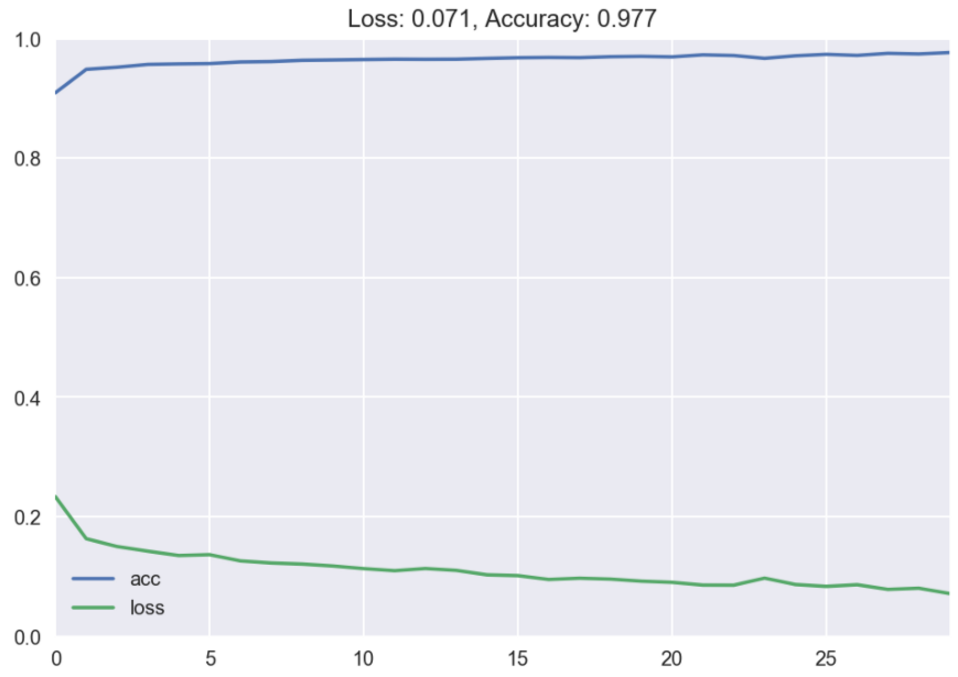
Now let’s build a deep neural network for binary classification. This model will be much more powerful, and will be able to model non-linear relationships.

```
deep_model = Sequential()
deep_model.add(Dense(64, input_shape=(X_train.shape[1],), activation='tanh'))
deep_model.add(Dense(16, activation='tanh'))
deep_model.add(Dense(1, activation='sigmoid'))

deep_model.compile(Adam(lr=0.01), 'binary_crossentropy', metrics=['accuracy'])

deep_history = deep_model.fit(X_train, y_train, verbose=0, epochs=30)
plot_loss_accuracy(deep_history)
```

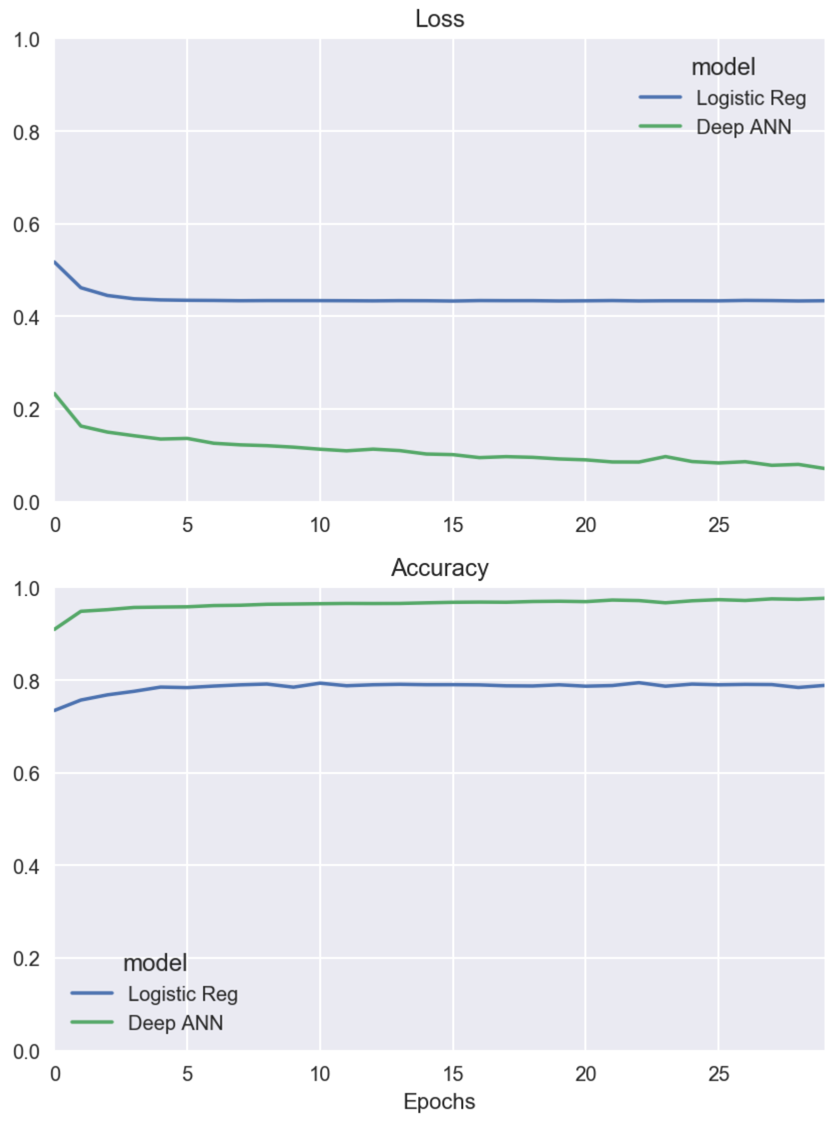
case_classification_7.py hosted with ♡ by GitHub view raw

The model building process is again very familiar. We have 2 hidden layers with 64 and 16 nodes with tanh activation function. The output layer uses the sigmoid activation since it's a binary classification problem. We use the Adam optimizer with learning rate set to 0.01.

This time we achieve 97.7% training accuracy, pretty good.

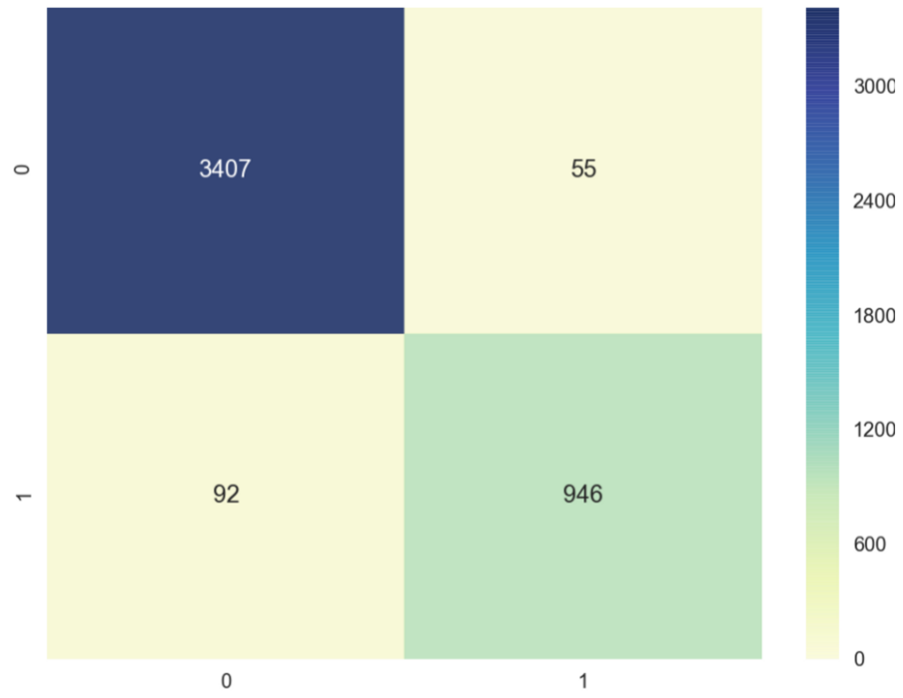


Let's compare the LR and ANN models. The ANN model is much superior, having a lower loss and a higher accuracy.



And for completeness here's the classification report and confusion matrix of the ANN model on the test set. We achieve 97% accuracy, compared to 78% of the LR model. We still misclassify 147 examples out of 4500.

```
             precision    recall  f1-score   support

          0       0.97      0.98      0.98      3462
          1       0.95      0.91      0.93      1038

avg / total       0.97      0.97      0.97      4500
```



We can further improve the performance of the ANN by doing the following:

- Train the model for longer (increase the number of epochs).
- Hyperparamter tuning: change the learning rate, use a different optimizer than Adam (RMSprop for example), use another activation

function than tanh (can be relu).

- Increase the number of nodes per layer: Instead of 64–16–1 we can do 128–64–1.
- Increase the number of layers: We can do 128–64–32–16–1.

One important caveat though, as we make the model more powerful, the training loss will likely decrease and accuracy will increase. But we will run into the risk of overfitting. Meaning the complex model will perform worse on the test set compared to a simpler model, even though the training metrics of the complex model is better. We will talk more about overfitting in another article, but this is very important to keep in mind. That's why we don't go crazy with number of layers and nodes per layer. The simplest model that gets the job done is sufficient.

1.4) Visualization of Deep ANN

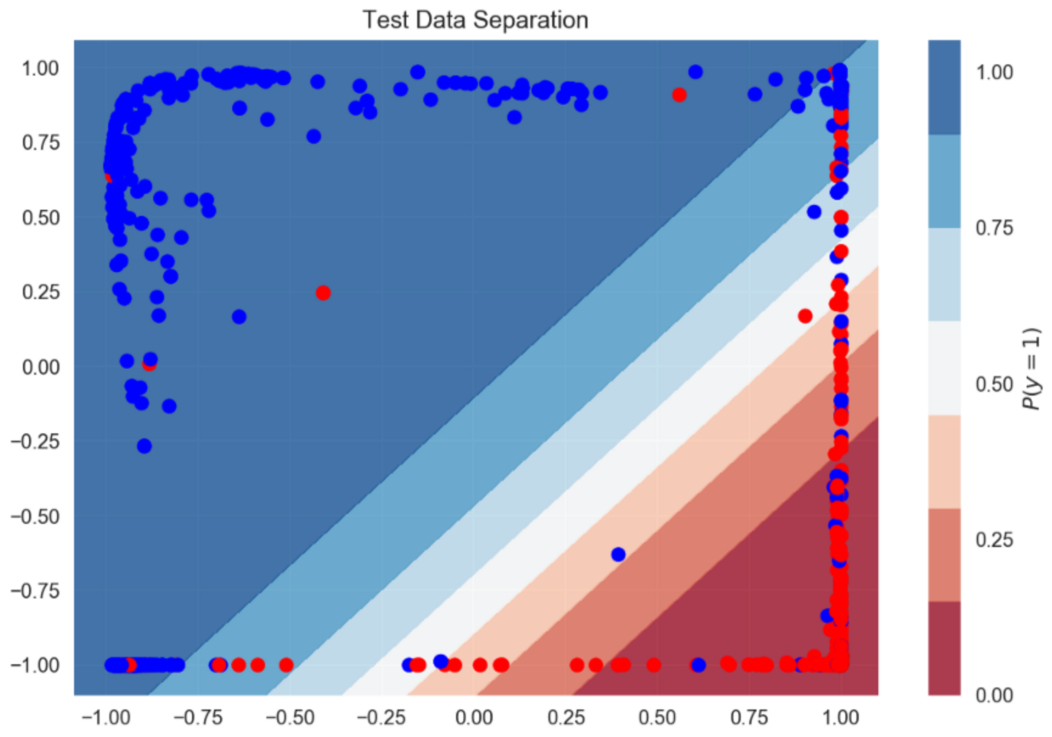
In the previous article we learned that each layer of the ANN performs a non-linear transformation of the input from one vector space to another. By doing this we are projecting our input data to a new space where the classes are separable from each other via a complex decision boundary.

Let's visually demonstrate this. Our input data after the initial data preprocessing we did above is 20 dimensional. For visualization purposes let's project it to 2D. Remember that having k nodes in a layer means that this layer transforms its input such that the output is a k-dimensional vector. The ANN we trained above had two hidden layers with 64 and 16 nodes. Then we need a new layer with 2 nodes in order to project our data to a 2D space. So we add this new layer just before the output node. The rest is completely untouched.

```python
deep_model_vis = Sequential()
deep_model_vis.add(Dense(64, input_shape=(X_train.shape[1],), activation='tanh'))
deep_model_vis.add(Dense(16, activation='tanh'))
# this is the new layer
deep_model_vis.add(Dense(2, activation='tanh'))
deep_model_vis.add(Dense(1, activation='sigmoid'))

deep_model_vis.compile(Adam(lr=0.01), 'binary_crossentropy', metrics=['accuracy'])

deep_model_vis.fit(X_train, y_train, verbose=0, epochs=10)
```

case_classification_8.py hosted with ♡ by GitHub view raw

Here's the resulting projection of our input data from 20D to 2D space. The decision boundary corresponds to the last layer of the ANN. The ANN was able to separate out the classes pretty well, with some misclassifications. A lot of data points overlap in 2D so we can't see them all, for reference the model misclassifies around 160 points out of 4500 (96% accuracy). We aren't concerned about accuracy with this model anyway, we are interested in the projection of a high-dimensional input to 2D. This is a neat little trick to visually demonstrate the result of the projections performed by the ANN.



A more principled visualization approach would be using *t-SNE*, which is a dimensionality reduction technique for visualizing high-dimensional data. Details available here.

2. Case Study: MultiClass Classification

We will now perform multiclass classification on the famous Iris dataset. It contains 3 classes of flowers with 50 examples each. There are a total of 4 features. So it's pretty small, but very popular. The data looks as follows.

sepal_length	sepal_width	petal_length	petal_width	label
4.40	3.00	1.30	0.20	setosa
5.50	3.50	1.30	0.20	setosa
4.90	3.00	1.40	0.20	setosa
5.10	2.50	3.00	1.10	versicolor
6.40	2.80	5.60	2.10	virginica

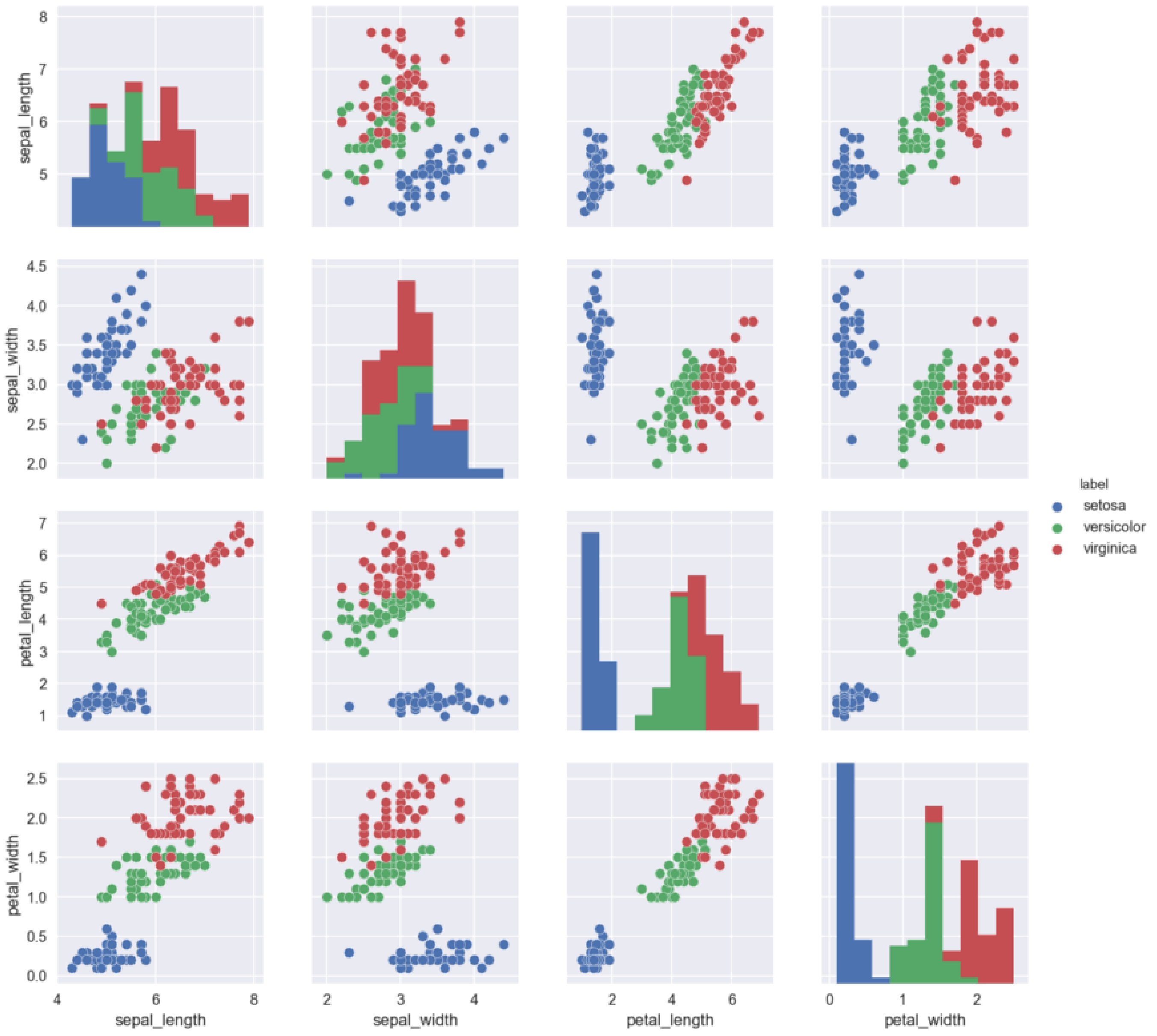
2.1) Data Visualization & Preprocessing

This part is easier now since we only have 4 real-valued features. Again we normalize the features to be between [0, 1]. The feature values are small and we could get away without any normalization, but it's a good habit to do so, and there's no harm doing it anyway.

Pairplot is a cool visualization technique if we have a small number of features. We can see the pairwise distribution of features colored by the class (the column "label" in the dataset). We use the *seaborn* package, pretty simple for an informative plot.

```
seaborn.pairplot(df, hue='label')
```

case_multiclass_1.py hosted with ♡ by GitHub view raw



We can see that the “setosa” class is easily separable but “versicolor” and “virginica” are more intermingled.

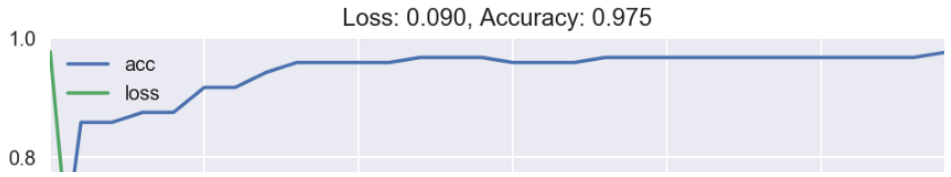
2.2) Softmax Regression Model

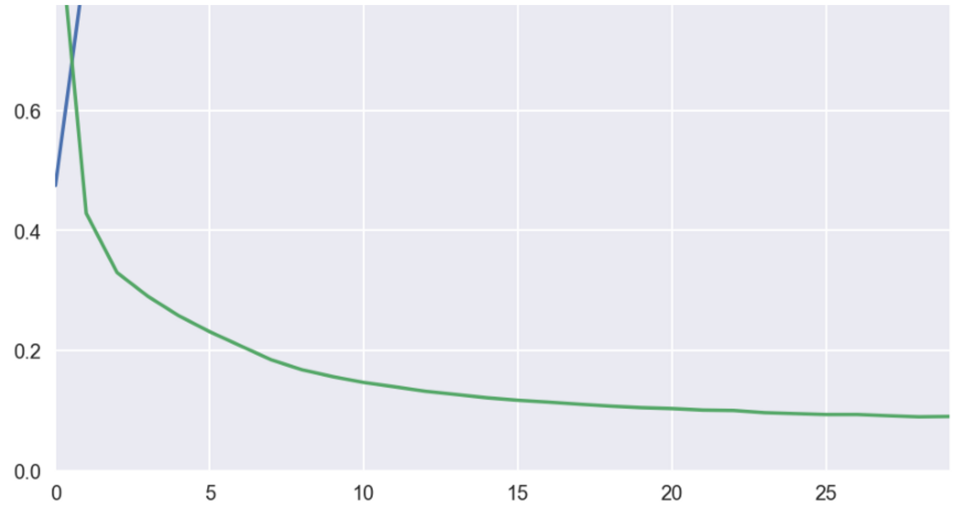
We will now train a Softmax Regression (SR) model to predict the labels. The previous article contains a detailed explanation, and building the model is again very similar to above. The main difference is that we’re using *softmax* activation and *categorical_crossentropy* as the loss.

```python
sr_model = Sequential()
sr_model.add(Dense(3, input_shape=(X.shape[1],), activation='softmax'))

sr_model.compile(Adam(lr=0.1), loss='categorical_crossentropy', metrics=['accuracy'])

history = sr_model.fit(X_train, y_train, epochs=30, verbose=0)
plot_loss_accuracy(history)
```

case_multiclass_2.py hosted with ♡ by GitHub view raw

The SR model already achieves 97% training accuracy and a minimal loss, pretty good already.





2.3) ANN Model

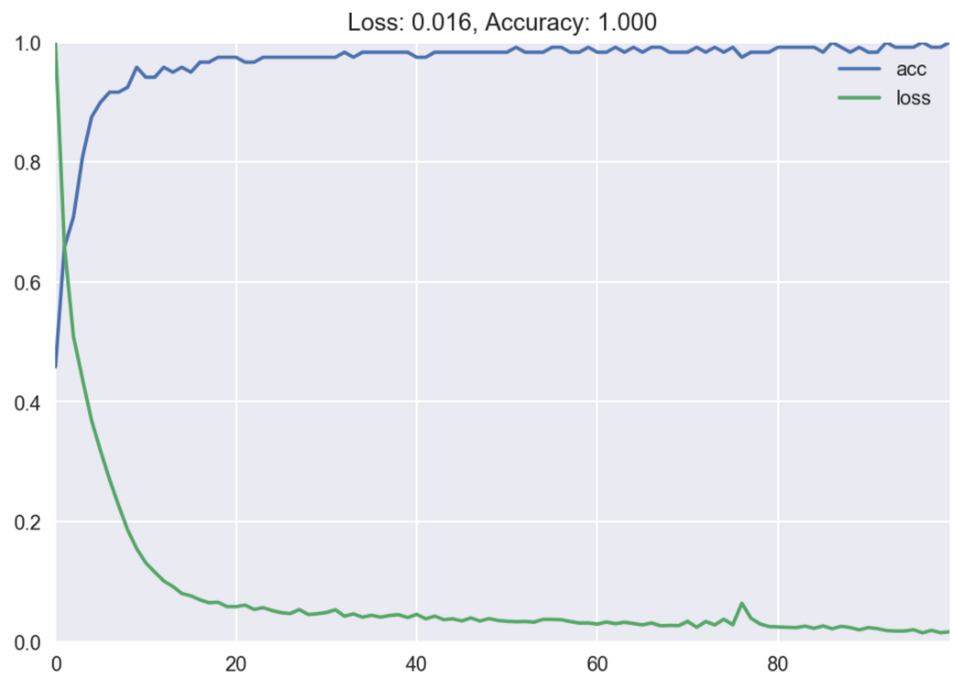
Now let's build our ANN model. We add 2 hidden layers with 32 and 16 nodes. Notice that we also change the activation function of these layers to *relu* instead of tanh. We will explore various activation functions and their differences in another tutorial, but relu is arguably the most popular one. Then why we haven't been using it? Well, only because the decision boundary plots looked prettier with tanh activation. Seriously, no other reason.

```
deep_model = Sequential()
deep_model.add(Dense(32, input_shape=(X.shape[1],), activation='relu'))
deep_model.add(Dense(16, input_shape=(X.shape[1],), activation='relu'))
deep_model.add(Dense(3, activation='softmax'))

deep_model.compile(Adam(lr=0.01), loss='categorical_crossentropy', metrics=['accuracy'])
```

```python

history = deep_model.fit(X_train, y_train, epochs=100, verbose=0)
plot_loss_accuracy(history)
```

case_multiclass_3.py hosted with ♡ by GitHub view raw

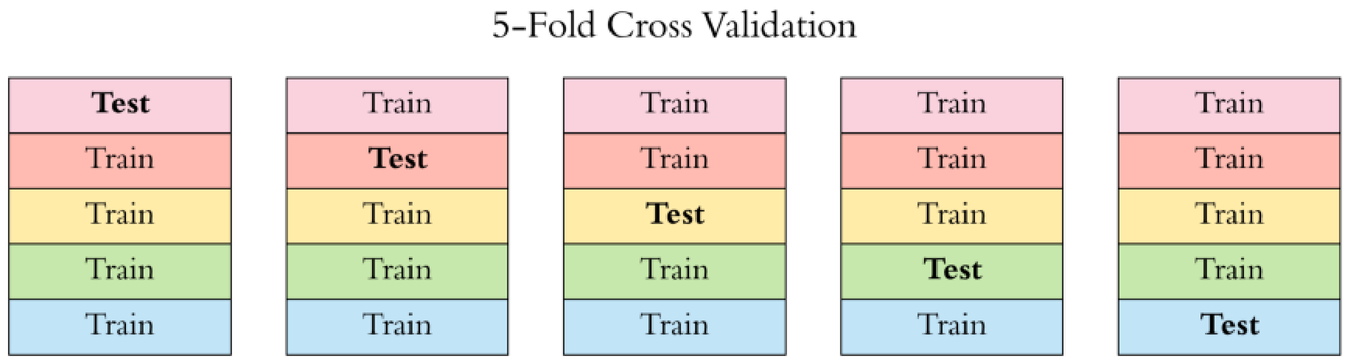
This time we get 100% traning accuracy.



And for the record, test accuracy of both SR and ANN models are 100%. This is a pretty small dataset, so these results are not surprising. Not all problems need a deep neural net to get good results. For this problem it's probably an overkill, since a linear model works just as fine.

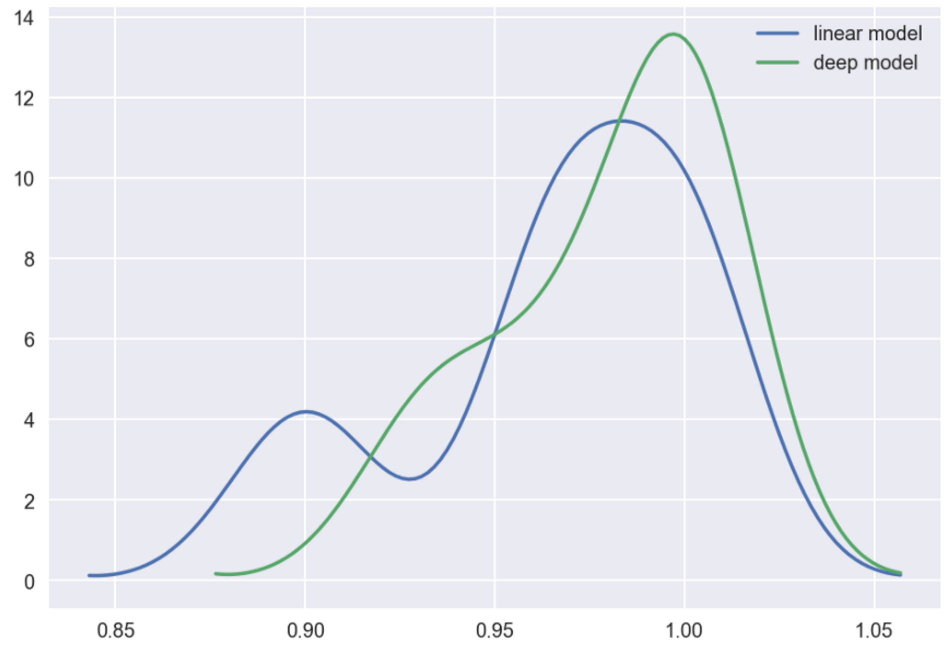
2.4) Cross Validation

With a small sample size like our current situation, it's especially important to perform cross validation to get a better estimate on accuracy. With *k-fold cross validation* we split the dataset into k disjoint parts, use k-1 parts for training and the other 1 part for testing. This way every example appears in both training and test sets. We then average out the model's performance in all k runs and get a better low-variance estimation of the model accuracy.



Usually while training deep learning models we don't perform k-fold cross validation. Because the training takes a long time, and training the model k times from scratch is not feasible. But since our dataset is small it's a good candidate to try on.

Here's the plot for 5-fold cross validation accuracy for both models. The deep model is doing slightly better, has a higher accuracy and lower variance. In the figure the accuracies sometimes seem to be above 100% but that's an artifact of smoothing the curves. Max accuracy we get is 100%.

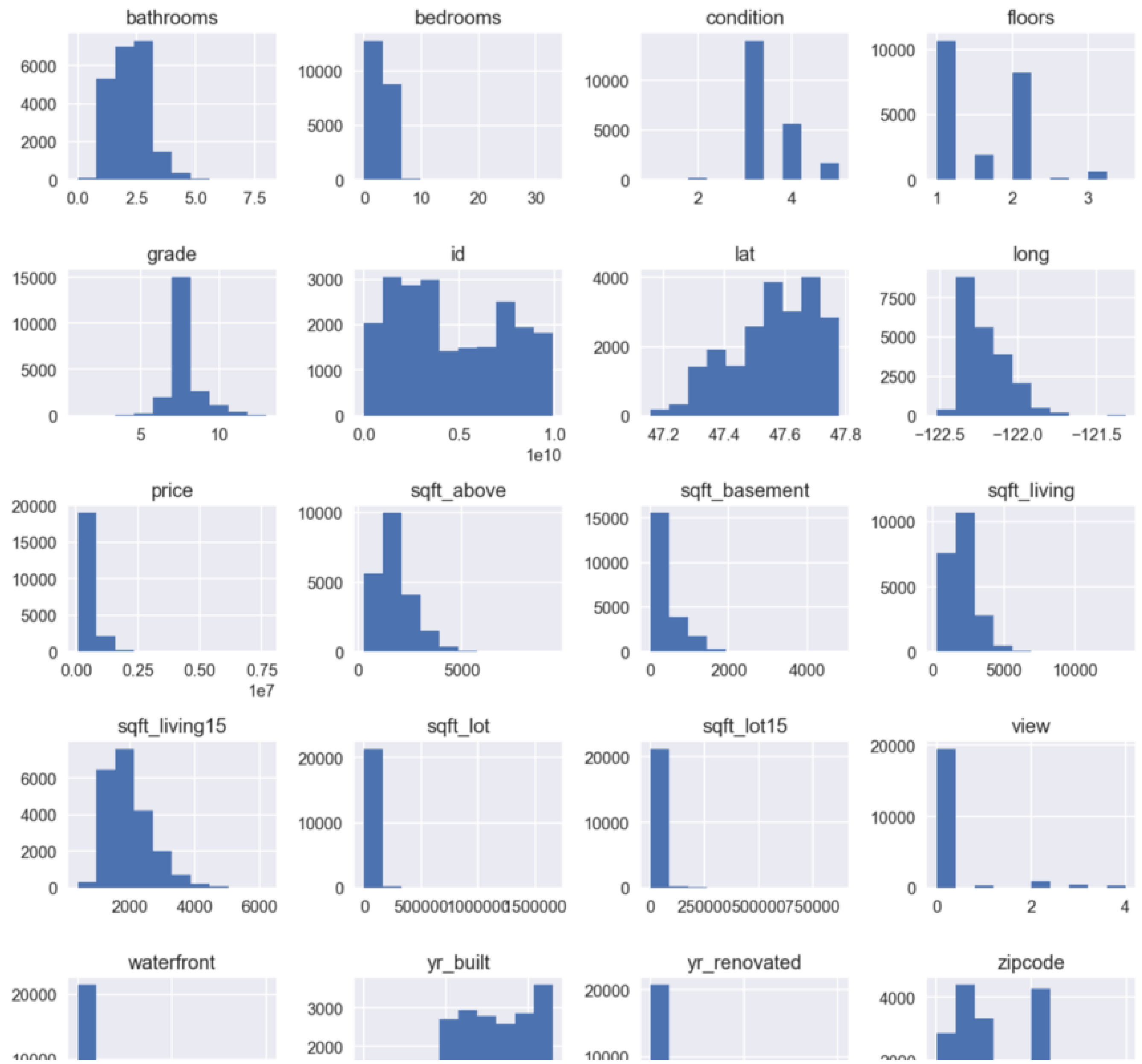


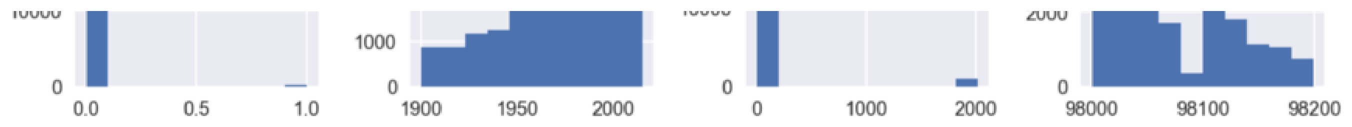
3. Case Study: Regression

We'll now work on a regression problem, predicting a real-valued output instead of discreet class memberships. We will be using the house sales dataset from King County, WA on Kaggle. There are around 21,000 rows with 20 features. The value we're trying to predict is a floating point number labeled as "price".

3.1) Data Visualization & Preprocessing

First let's take a look at feature distributions





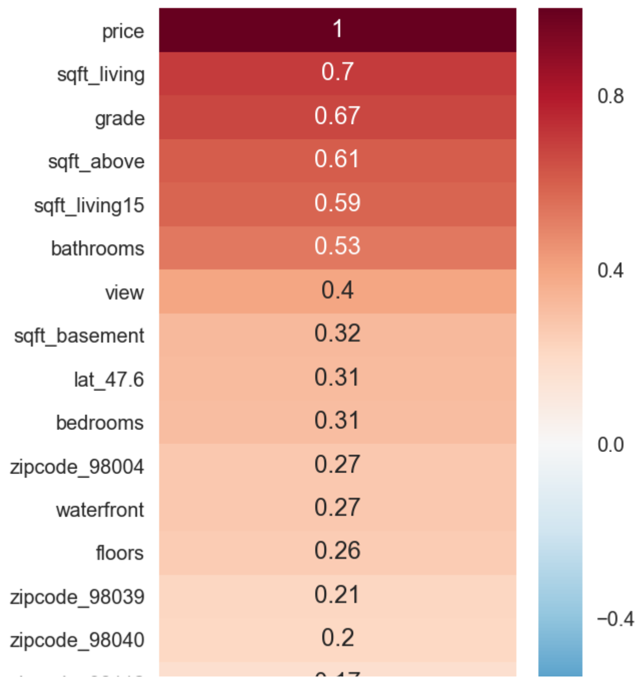
You know the drill by now, we need to do feature normalization and categorization. For example squarefoot related features definitely need to be normalized since the values range in thousands, and features like zipcode need to be categorized.

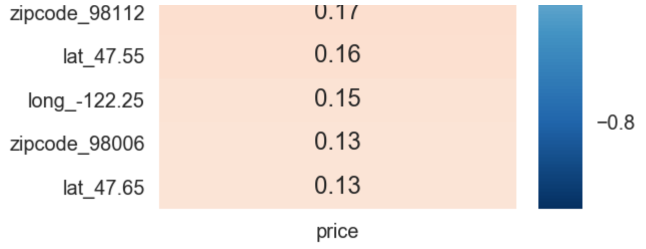
We also have a new type of preprocessing to do, *bucketization*. For example the feature which contains the year the house was built (yr_built), ranges from 1900 to 2015. We can certainly categorize it with every year belonging to a distinct category, but then it would be pretty sparse. We would get more signal if we bucketized this feature without losing much information. For example if we use 10 year buckets, years between [1950, 1959] would be collapsed together. It would probably be sufficient to know that this house was built in 1950s instead of 1958 exactly.

Other features that would benefit from bucketizing are latitude and longitude of the house. The exact coordinate doesn't matter that much, we can round the coordinate to the nearest kilometer. This way the feature values will be more dense and informative. There's no hard and set rule to which ranges to use in bucketization, they're mostly decided by trial and error.

One final transformation we need to do is for the price of the house, the value we're trying to predict. Currently its value ranges from $75K to $7.7M. A model trying to predict in such a large scale and variance would be very unstable. So we normalize that as well. Feel free to check the code for the details.

After all the transformations we go from 20 to 165 features. Let's check the correlation of each feature with price.





The most correlated feature is the square footage, which is expected, bigger houses are usually more expensive. Looking at the list the features make sense. Some zipcodes have high correlation with price, for example 98039 which corresponds to Medina, that's where Bill Gates lives and it's one of the most expensive neighborhoods in the country. There's another zipcode 98004 which is more correlated corresponding to Bellevue. There are a lot of high-rises and tech offices there, which has been driving up the prices a lot lately. I used to live in that neighborhood, but then it got too boring and expensive so I moved :)

3.2) Linear Regression Model

This is the first time we're building a regression model. Just like Logistic Regression is the simplest model we try first in a classification problem, Linear Regression is the one we start with in a regression problem.

Remember that the equation for logistic regression is $y=f(xW)$ where f is the sigmoid function. Linear regression is simply $y=xW$, that's it no activation function. I've again omitted the bias terms for simplicity. With the biases they become *y=f(xW+b)* and *y=xW+b* respectively.

As a reminder here is how we built the logistic regression (LR) model

```
lr_model = Sequential()
lr_model.add(Dense(1, input_shape=(X_train.shape[1],), activation='sigmoid'))

lr_model.compile(Adam(lr=0.01), 'binary_crossentropy', metrics=['accuracy'])

lr_history = lr_model.fit(X_train, y_train, verbose=1, epochs=30)
plot_loss_accuracy(lr_history)
```

case_classification_6.py hosted with ♡ by GitHub view raw

And here's the code for the linear regression model (LinR)

```
linr_model = Sequential()
linr_model.add(Dense(1, input_shape=(X.shape[1],)))

linr_model.compile('adam', 'mean_squared_error')

linr_history = linr_model.fit(X_train, y_train, epochs=30, validation_split=0.2)
plot_loss(linr_history)
```

case_regression_1.py hosted with ♡ by GitHub view raw

There are 3 main differences:

- LR uses a sigmoid activation function, where LinR has no activation.
- LR uses binary_crossentropy loss function, where LinR uses mean_squared_error.
- LR also reports the accuracy, but accuracy is not an applicable metric to a regression problem, since the output is a floating point number instead of a class membership.

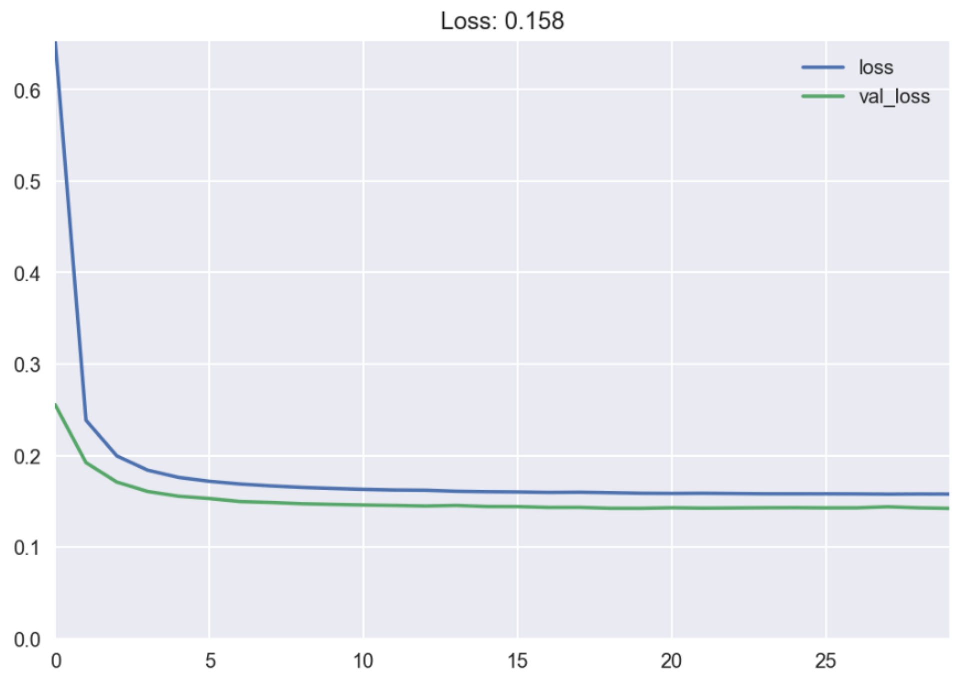
The most important change is the loss function *mean_squared_error* (MSE). MSE is a standard loss function used for regression. The formula is pretty simple:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2$$

Where *y* is the true value, *ŷ* is the predicted and *n* is the number of examples.

We’re also passing a new *validation_split* argument to the fit function. It specifies the fraction of training data to use as held-out validation set during training, in our case it’s 20%. Using a validation set we can see whether we’re overfitting during training. Don’t confuse the validation set with the test set though. Test set is entirely separate and it doesn’t get exposed to the model at all during training.

The loss decreases in the first couple of epochs and then stabilizes. We’re likely *underfitting* meaning our model doesn’t have enough capacity, and we need a more compex model.



And these are the top 10 features with the highest weights. Categorical zipcode features are highly dominant.

	feature_weight
zipcode_98039	1.87
waterfront	1.46
zipcode_98004	1.17
zipcode_98112	0.76
zipcode_98040	0.61
zipcode_98109	0.48
zipcode_98105	0.44
zipcode_98119	0.42
zipcode_98102	0.41
lat_47.6	0.39

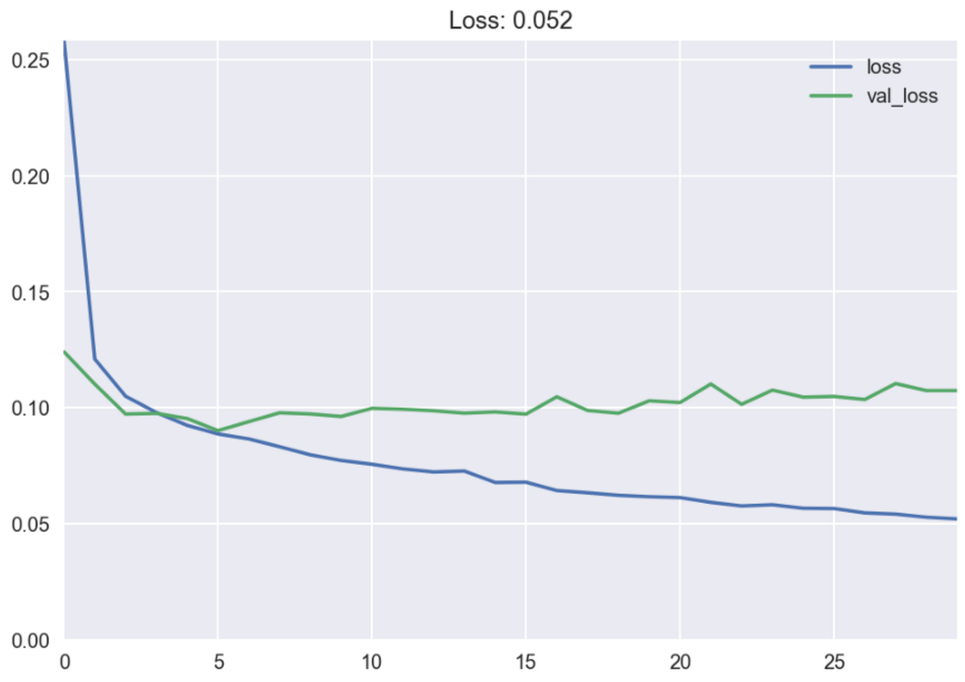
3.3) ANN Model

And finally let's build an ANN for regression. In the previous examples, going from a linear model to a deep model just involved adding new layers with non-linear activation functions. It will be the same this time as well.

```
deep_model = Sequential()
deep_model.add(Dense(32, input_shape=(X.shape[1],), activation='relu'))
deep_model.add(Dense(16, activation='relu'))
deep_model.add(Dense(8, activation='relu'))
deep_model.add(Dense(1))

deep_model.compile('adam', 'mean_squared_error')

deep_history = deep_model.fit(X_train, y_train, epochs=30, validation_split=0.2)
plot_loss(deep_history)
```

case_regression_2.py hosted with ♡ by GitHub view raw

We added new layers with relu activation. The loss plot looks interesting now. The training error loss still seems to be decreasing, but the validation error starts increasing after the 5th epoch. We're clearly *overfitting*. The ANN is memorizing the training data, and this is reducing its ability to generalized on the validation set.

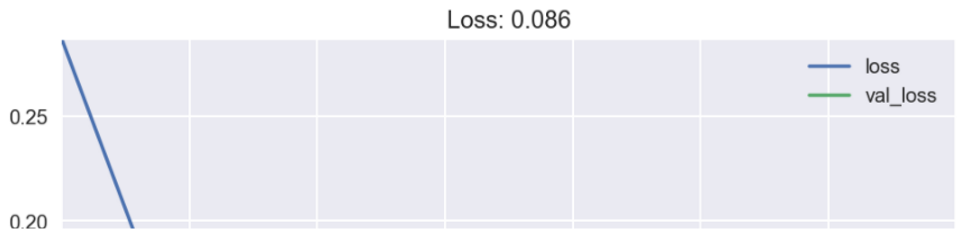


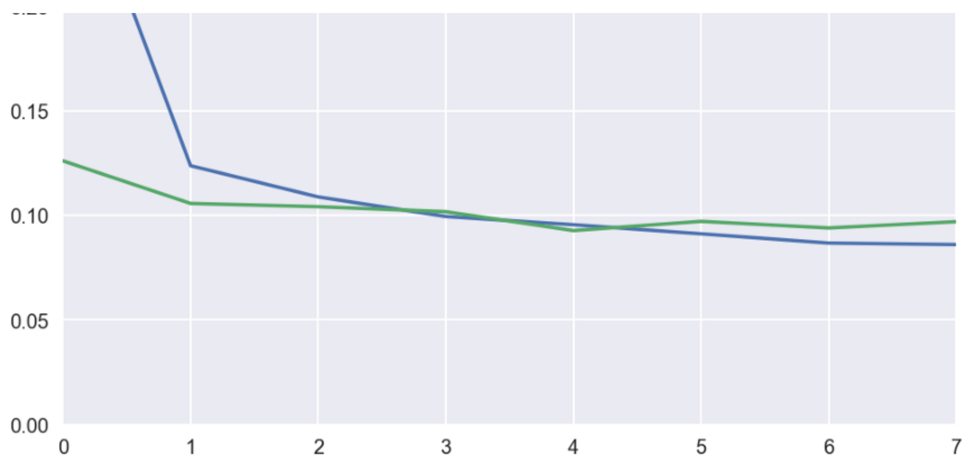
The solution? There are a couple of methods for tackling overfitting in deep neural nets. Most of the methods rely on constraining the capacity of the model. This can be achieved for example by restricting weights, sharing weights or even stopping training before the training loss plateaus. We will use the last one in the interest of brevity and discuss the rest in another article. This means we will simply halt training as soon as the validation loss stops improving. This is called *early stopping*, and it's pretty easy to implement with Keras. We just need to modify the call to the fit function as follows. If there's no improvement for validation loss for 2 epochs, we stop training.

```
early_stop = EarlyStopping(monitor='val_loss', patience=2, verbose=1)
deep_history = deep_model.fit(X_train, y_train, epochs=30, validation_split=0.2,
                              callbacks=[early_stop])
```

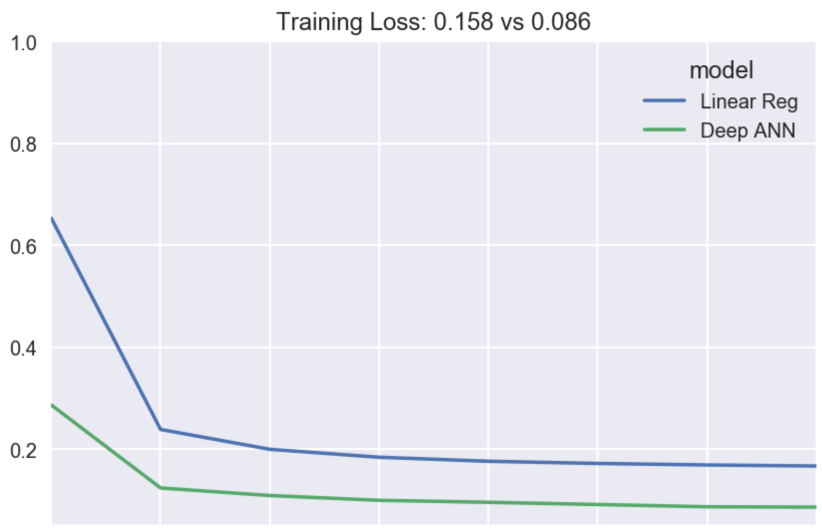
case_regression_3.py hosted with ♡ by GitHub view raw

The validation loss stopped improving at the 5th epoch, the model waited for 2 more epochs and finished training at epoch 7.

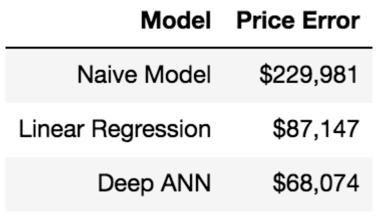




The training loss with the LinR model was 0.158, the loss for ANN is 0.086. We got 83% improvement over the linear model, pretty good. And comparing the loss on the test set, LinR model got 0.191 compared to 0.127 of ANN which is 32% improvement. The figure below compares the training loss of LinR vs ANN.



Let’s make a final comparison, dollar value difference between the model predicted price and actual price. The most naive model which always predicts the average price of the training set ($540K), is off by $229K on the test set, pretty bad. Linear Regression model is off by $87K vs $68K for the Deep ANN . The ANN is 21% better than LinR.



Model	Price Error
Naive Model	$229,981
Linear Regression	$87,147
Deep ANN	$68,074

4) Conclusion

I hope this was an informative article. I tried to demonstrate a step-by-step application of deep learning on 3 common machine learning problems with real-life datasets.

Data preprocessing and visualization was covered in detail. Even though they’re not the fun part of solving an ML problem and mostly overlooked, they're extremely important. Just like Part 1 we first tackled the problem with a simple model, and then used a deep neural net to get better results.

The entire code for this article is available here if you want to hack on it yourself. If you have any feedback feel free to reach out to me on twitter.

Machine Learning Deep Learning Neural Networks Data Science Towards Data Science