An input layer can be imagined as a "gate" to the model. The gate has a defined shape. This shape should be consistent with the input sample. This layer has two functions,

- not allow a sample to get through if its shape is not consistent, and
- communicate the shape of the inputted batch to the next layer.

Listing 4.7. Input layer in a Sequential model.

```
model.add(Input(shape=(N_FEATURES, )))
```

The input layer is added to the `model` in Listing 4.7. It takes an argument `shape` which is a tuple of the shape of the input. The tuple contains the length of each axes of the input sample and an empty last element.

Here the input has only one axis for the features (shown in Figure 4.2) with a length `N_FEATURES` defined in Listing 4.3. In the case of multi-axes inputs, such as images and videos, the tuple will have more elements.

The last (empty) element corresponds to the batch size. The batch size is defined during the model fit and is automatically taken by the model. The empty element in the tuple can be seen as a placeholder.

Explicitly defining the input layer is optional. In fact, it is common to define the input shape in the first computation layer, e.g. as `Dense(..., input_shape=(N_FEATURES, ))`. It is still explicitly defined here for clarity.

4.4.3 Dense Layer

A dense layer is one of the primary layers in deep learning. It is used in MLPs and most other deep learning architectures.

Its primality can be attributed to its simplicity. A linearly activated dense layer is simply an *affine* transformation of the inputs. It will be explained in the future chapters that such affine transformations make model construction for different architectures, such as LSTM autoencoder, easier.

Moreover, as opposed to most other layers, a (non)linear dense layer provides a simple structure to find a relationship between the features and response in the same space.

An MLP is a stack of dense layers. That is, from hidden to output all are dense layers[9].

The number of hidden layers is a model configuration. As a rule-of-thumb, it is recommended to begin with two hidden layers as a baseline. They are added to the model in Listing 4.8.

Listing 4.8. Hidden Layers in a Sequential model.

```
model.add(Dense(32, activation='relu', name='
    hidden_layer_1'))
model.add(Dense(16, activation='relu', name='
    hidden_layer_2'))
```

The size of a layer is the first argument. The number of nodes (denoted as `units` in TensorFlow) in the layer is the same as its size (see Figure 4.2).

The size is a configuration property. It should be set around half of the number of input features. As a convention the size should be taken from a geometric series of 2: a number in $\{1, 2, 4, 8, 16, 32, \ldots\}$.

The input sample has 69 features, therefore, the first dense layer is made of size 32. This also means the input to the second layer has 32 features, and thus, its size is set as 16.

Following these conventions are optional but help in streamlined model construction. These conventions are made keeping in account the insensitivity of deep learning models towards minor configuration changes.

Deep learning models are generally insensitive to minor changes in a layer size. Therefore, it is easier to follow a rule-of-thumb for configuring layer sizes.

Activation is the next argument. This is an important argument

[9] An MLP with only **one** dense layer (the output layer) is the same as a logistic regression model.

as the model is generally sensitive to any ill selection of activation. `relu` is usually a good first choice for the hidden layers.

Appropriate choice of activation is essential because models are sensitive to them. ***`Relu`*** *activation is a good default choice for hidden layers.*

The `name` argument, in the end, is optional. It is added for better readability in the *model summary* shown in Figure 4.4. And, if needed, to draw layer attributes, such as layer weights, using the name property.

4.4.4 Output Layer

The output layer in most deep learning networks is a dense layer. This is due to dense layer's *affine* transformation property which is usually required at the last layer. In an MLP, it is a dense layer by design.

The output layer should be consistent with the response's size just like the input layer must be consistent with the input sample's size.

In a classification problem, the size of the output layer is equal to the number of classes/responses. Therefore, the output dense layer has a unit size in a binary classifier (`size=1`) in Listing 4.9.

Listing 4.9. Output Layer in a Sequential model.

```
model.add(Dense(1, activation='sigmoid', name='
    output_layer'))
```

Also, the activation on this layer is dictated by the problem. For regression, if the response is in $(-\infty, +\infty)$ the activation is set as `linear`. In binary classifiers it is `sigmoid`; and `softmax` for multi-class classifiers.

4.4.5 Model Summary

At this stage, the model network has been constructed. The model has layers from input to output with hidden layers in between. It is useful to visualize the summary of the network.

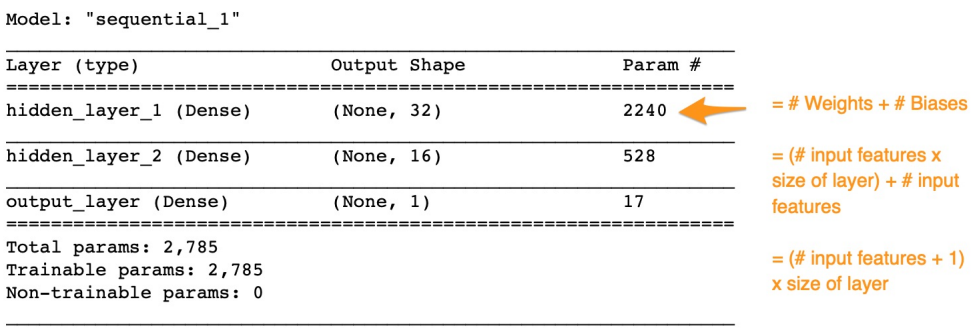


Figure 4.4. *MLP Baseline Model Summary.*

Listing 4.10 displays the model summary in a tabular format shown in Figure 4.4.

Listing 4.10. Model Summary.

```
model.summary()
```

In the summary, `Model: "sequential_‘x’"` tells this is a `Sequential` TensorFlow model object. The layers in the network are displayed in the same order as they were added.

Most importantly, the shape of each layer and the corresponding number of parameters are visible. The number of parameters in a dense layer can be derived as: *weights* = *input features* × *size of layer* plus *biases* = *input features*. However, a direct visibility of the layer-wise and an overall number of parameters help to gauge the shallowness or massiveness of the model.

The end of the summary shows the total number of parameters and its breakdown as *trainable* and *non-trainable* parameters. All the weight and bias parameters on the layers are trainable parameters. They are trained (iteratively estimated) during the model training.

Examples of non-trainable parameters are the network topology configurations, such as the number of hidden layers and their sizes. Few other non-trainable parameters are in layers such as `Dropout` and `BatchNormalization`. A `Dropout` parameter is preset while `BatchNormalization` parameters, e.g., the batch mean and variance,

are not "trained" but derived during the estimation.

In fact, in some cases, an otherwise trainable `Dense` layer is deliberately made non-trainable by
`model.get_layer(layerName).trainable = False`. For example, upon using a pre-trained model and training only the last layer.

However, in most models like here, all the parameters are trainable by default.

4.4.6 Model Compile

So far, the model is constructed and visually inspected in the model summary. This is like setting the layers of a cake on a baking tray and inspecting it before placing it in the oven. But before it goes in the oven there is an oven configuration step: set the right mode and temperature.

Similarly, in TensorFlow, before the model is fed in the machine for training there is a configuration step called *model compilation*.

This is done using `model.compile` in Listing 4.11.

Listing 4.11. Model Compile.

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy',
                       tf.keras.metrics.Recall(),
                       performancemetrics.F1Score(),
                       performancemetrics.
                          FalsePositiveRate()]
                )
```

The `model.compile` function has two purposes. First, verify any network architecture flaws, such as inconsistencies in input and output of successive layers. An error is thrown if there is any inconsistency.

And, second, define the three primary arguments: `optimizer`, `loss`, and `metrics` (there are other optional arguments not covered here). Each of these essential arguments is explained in the following.

- `optimizer`. A variety of optimizers are available with Tensor-

Flow[10]. Among them, `adam` is a good first choice. Quoted from its authors in Kingma and Ba 2014,

> "(Adam) is designed to combine the advantages of two recently popular methods: AdaGrad (Duchi et al., 2011 Duchi, Hazan, and Singer 2011), which works well with sparse gradients, and RMSProp (Tieleman & Hinton, 2012 G. Hinton et al. 2012), which works well in on-line and non-stationary settings..."

Almost every optimizer developed for deep learning are gradient-based. *Adam*, among them, is computationally and memory efficient. Moreover, it is less sensitive to its hyper-parameters. These attributes make *adam* an appropriate choice for deep learning.

The original adam paper (Kingma and Ba 2014) is simple-to-read and is recommended for more details.

`Adam` is a robust optimizer and, hence, a default choice.

- `loss`. The choice of a loss function depends on the problem. For example, in regression `mse` is a good choice while `binary_crossentropy` and `categorical_crossentropy` work for binary and multi-class classifiers, respectively.

  A list of loss functions for different needs are provided by TensorFlow[11]. For the binary classification problem here, `binary_crossentropy` loss is taken. This loss function is shown in Equation 2.2 and succinctly expressed again as

$$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\Big(y_i \log(p_i) + (1-y_i)\log(1-p_i)\Big) \tag{4.7}$$

[10] `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/optimizers`

[11] `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/losses`

where $y_i \in 0, 1$ are the true labels and $p_i = \Pr[y_i = 1]$ is the predicted probability for $y_i = 1$. Looking closely at the equation, we can see that the loss is minimum when there are perfect predictions, i.e., if $p_i \to 1 | y_i = 1$ **and** $p_i \to 0 | y_i = 0$.

Also, it can be noted in Equation 4.7 that the cross-entropy loss is continuous and differentiable. Thus, it works well with the gradient descent based backpropagation estimation methods.

Cross entropy loss is continuous and differentiable and, hence, works well with gradient-based backpropagation method in deep learning.

- `metrics`. Unlike the `optimizer` and `loss`, metrics are not directly related to model training. Rather, the metrics are for evaluating the model performance during training and inferencing.

  Metrics are unrelated to training because their values, whether good or poor, do not impact the model parameter update during the training. The training focuses only on optimizing (generally minimizing) the loss function. The metrics are an outcome of this process.

  In regression, `RootMeanSquaredError()` is an appropriate metric. In classification, the `accuracy` that shows the percent of correct classification is generally used. However, for imbalanced data sets the objective is to have a high *recall* and *f1-score*, and a low *false positives rate* (fpr) (described in § 2.1.3).

  TensorFlow provides an in-built definition for recall (`Recall()`) but falls shy for the latter two. Therefore, custom f1-score and fpr metrics are constructed and elaborated in § 4.8.2. Also, the inability to use outside functions becomes clearer there.

The model compilation is the first stage-gate towards model completion. A successful model compile is a moment to cherish before heading to the next step: model fit.

4.4.7 Model Fit

Model fitting is a step towards the "moment of truth." This is the time when the model performance is seen.

Listing 4.12. Model Fit.

```
history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                        y_valid),
                    verbose=1).history
```

The primary arguments in `model.fit` is `x`, `y`, `batch_size`, and `epochs`.

- `x`: It is the training features X in a numpy array or pandas data frame. It is recommended to be scaled. In this model, `StandardScaler` scaled features are used.

- `y`: It is the response $\boldsymbol{y}$ aligned with the features X, i.e., the i-th row in X should correspond to the i-th element in $\boldsymbol{y}$. It is a one-dimensional array for a single class (response) model. But `y` can be a multi-dimensional array for a multi-class (response) model.

- `batch_size`: The model trains its parameters batch-by-batch using a batch gradient descent based approach. This approach is an extension of stochastic gradient descent (SGD) which uses only one sample at a time. While the batch size can be set to one like in SGD, deep learning optimization with one sample at a time is noisy and can take longer to converge. Batch gradient descent overcomes this issue while still having the benefits of SGD.

  Similar to the hidden layer sizes, the model is not extremely sensitive to minor differences in batch size. Therefore, it is easier to choose the batch size from the geometric series of 2. For balanced data sets, it is suitable to have a small batch, e.g., 16. However, for imbalanced data sets, it is recommended to have a larger batch size. Here it is taken as 128.

- `epochs`: Epochs is the number of iterations of model fitting. The number of epochs should be set such that it ensures model convergence. One indication of model convergence is a stable loss, i.e., no significant variations or oscillations. In practice, it is recommended to start with a small epoch for testing and then increase it. For example, here the epochs were set as 5 first and then made 100. The model training can also be resumed from where it was left by rerunning `model.fit`.

If `model.fit` *is rerun without re-initializing or redefining the model, it starts from where the previous fit left. Meaning, the model can be incrementally trained over separate multiple runs of* `model.fit`.

An additional but optional argument is `validation_data`. Validation data sets can be either passed here or an optional argument `validation_split` can be set to a float between 0 and 1. Although validation is not mandatory for model fitting, it is a good practice. Here, the validation data sets are passed through the `validation_data` argument.

Listing 4.12 executes the model fit. The `fit()` function returns a list `history` containing the model training outputs in each epoch. It contains the values for the loss, and all the metrics mentioned in `model.compile()`. This list is used to visualize the model performance in the next section.

4.4.8 Results Visualization

Visualizing the results is a natural next step after model fitting. Visualizations are made for,

- **loss**. The progression of loss over the epochs tells about the model convergence. A stable and decreasing loss for the training data shows the model is converging.

  The model can be assumed to have converged if the loss is not changing significantly towards the ending epochs. If it is still de-

creasing, the model should be trained with more epochs.

An oscillating loss, on the other hand, indicates the training is possibly stuck in local minima. In such a case, a reduction in the optimizer learning rate and/or a change in batch size should be tried.

A stabilized decreasing loss is also desirable in the validation data. It shows the model is robust and not overfitted.

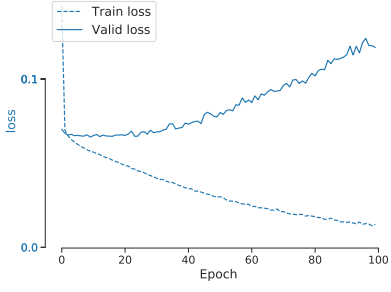
- **metrics**. Accuracy metrics on the validation data is more relevant than training. The accuracy-related metrics should improve as the model training progresses through the epochs. This is visualized in the metrics' plots.

  If the metrics have not stabilized and still improving, the model should be trained with more epochs. Worse, if the metrics are deteriorating with the epochs then the model should be diagnosed.

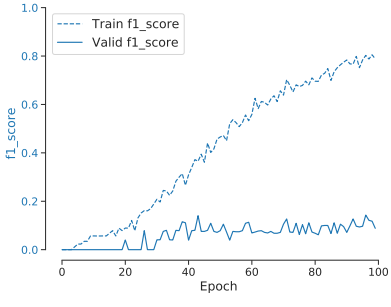
A few custom plotting functions are defined in Appendix C to visualize the results. Using them the results are plotted (and saved to a file) in Listing 4.13.

Listing 4.13. MLP baseline model results.

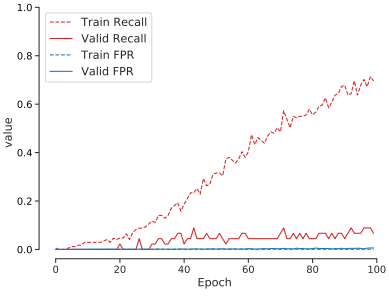
```
# Plotting loss and saving to a file
plt, fig =
  simpleplots.plot_metric(history,
    metric='loss')
fig.savefig('mlp_baseline_loss.pdf',
  bbox_inches='tight')

# Plotting f1score and saving to a file
plt, fig =
  simpleplots.plot_metric(history,
    metric='f1_score', ylim=[0., 1.])
fig.savefig('mlp_baseline_f1_score.pdf',
  bbox_inches='tight')

# Plotting recall and fpr, and saving to a file
plt, fig =
  simpleplots.plot_model_recall_fpr(history)
fig.savefig('mlp_baseline_recall_fpr.pdf',
  bbox_inches='tight')
```



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 4.5. *MLP baseline model results.*

The loss is shown in Figure 4.5a. The loss is stable and almost plateaued for the training data. However, the validation loss is increasing indicating possible overfitting.

Still the performance metrics of the validation set in Figure 4.5b-4.5c are reasonable. F1-score is more than 10%. The recall is around the same and the false positive rate is close to zero.

Completing a baseline model is a major milestone. As mentioned before, the multi-layer perceptron is the "hello world" to deep learning. A fair performance of the baseline MLP shows there are some predictive patterns in the data. This is a telling that with model improvement and different networks a usable predictive model can be built.

The next section will attempt at some model improvements for multi-layer perceptrons while the upcoming chapters will construct more in-

tricate networks, such as recurrent and convolutional neural networks.

4.5 Dropout

A major shortcoming of the baseline model was overfitting. Overfitting is commonly due to a phenomenon found in large models called **co-adaptation**. This can be addressed with **dropout**. Both, the co-adaptation issue and its resolution with dropout, are explained below.

4.5.1 What is Co-Adaptation?

If all the weights in a deep learning network are learned together, it is usual that some of the nodes have more predictive capability than the others.

In such a scenario, as the network is trained iteratively these powerful (predictive) nodes start to suppress the weaker ones. These nodes usually constitute a fraction of all. But over many iterations, only these powerful nodes are trained. And the rest stop participating.

This phenomenon is called co-adaptation. It is difficult to prevent with the traditional $\mathcal{L}_1$ and $\mathcal{L}_2$ regularization. The reason is that they also regularize based on the predictive capability of the nodes. As a result, the traditional methods become close to deterministic in choosing and rejecting weights. And, thus, a strong node gets stronger and the weak get weaker.

A major fallout of co-adaptation is: expanding the neural network size does not help.

If co-adaptation is severe, enlarging the model does not add to learning more patterns. Consequently, neural networks' size and, thus, capability get limited.

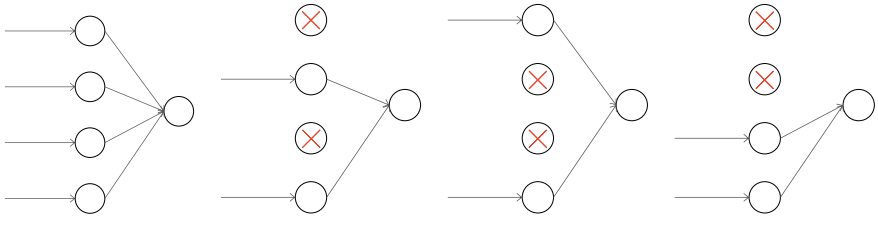
This had been a serious issue in deep learning for a long time. Then, in around 2012, the idea of *dropout*—a new regularization approach—emerged.

Dropout resolved co-adaptation. And naturally, this revolutionized deep learning. With dropout, deeper and wider networks were possible.

One of the drivers for the deep learning successes experienced today is attributed to dropout.

4.5.2 What Is Dropout?

Dropout changed the approach of learning weights. Instead of learning all the network weights together, dropout trains a subset of them in a batch training iteration.



(a) *Learning all weights.* (b) *Learning a subset of all weights with dropout.*

Figure 4.6. *Illustrating the difference in weight learning with and without Dropout. Without dropout (left), weights on all the connections are learned together. Due to this, if a few connections are stronger than others (because of initialization or the nature of the data) the other connections become dormant. With dropout (right), connections are randomly dropped during each training iteration. As a result, no connection gets a chance to maintain dominance. Consequently, all the weights are appropriately learned.*

Figure 4.6a-4.6b illustrates the model weights training (updates) during a batch iteration. Here a simple example to train weights of four nodes is shown. The usual training without dropout is in Figure 4.6a. In this, all the nodes are active. That is, all the weights will be trained together.

On the other hand, with dropout, only a subset of nodes are kept active during batch learning. The three illustrations in Figure 4.6b correspond to three different batch iterations (refer to the iteration levels in Figure 4.14). In each batch iteration, half of the nodes are switched off

while the remaining are learned. After iterating through all the batches the weights are returned as the average of their batch-wise estimations.

This technique acts as network regularization. But familiarity with traditional methods might make dropout appear not a regularization. Yet, there are some commonalities.

Like $\mathcal{L}_1$ regularization pushes the small weights to zero, dropout pushes a set of weights to zero. Still, there is an apparent difference: $\mathcal{L}_1$ does a data-driven suppression of weights while dropout does it at random.

Nevertheless, dropout is a regularization technique. It is closer to an $\mathcal{L}_2$ regularization. This is shown mathematically by Pierre and Peter in Baldi and Sadowski 2013. They show that under linearity (activation) assumptions the loss function with dropout (Equation 4.8 below) has the same form as $\mathcal{L}_2$ regularization.

$$\mathcal{L} = \frac{1}{2}\Big(t - (1-p)\sum_{i=1}^{n} w_i x_i\Big)^2 + \underbrace{p(1-p)\sum_{i=1}^{n} w_i^2 x_i^2}_{\text{Regularization term.}} \tag{4.8}$$

where p is the dropout rate.

The dropout rate is the fraction of nodes that are dropped at a batch iteration. For example, p is 0.5 in the illustration in Figure 4.6b.

The regularization term in Equation 4.8 has a penalty factor $p(1-p)$. The factor $p(1-p)$ is maximum when $p = 0.5$. Therefore, the dropout regularization is the largest at $p = 0.5$.

Dropout is a regularization technique equivalent to $\mathcal{L}_2$ regularization under linearity assumptions.

A dropout rate $p = 0.5$ is an ideal choice for maximum regularization.

Therefore, a dropout rate of 0.5 is usually a good choice for hidden

layers. If a model's performance deteriorate with this dropout rate it is usually better to increase the layer size instead of reducing the rate.

Dropout, if applied on the input layer, should be kept small. A rule-of-thumb is 0.2 or less. Dropout at the input layer means not letting a subset of the input features into the training iteration. Although arbitrarily dropping features is practiced in a few bagging methods in machine learning, it generally does not bring significant benefit to deep learning. And, therefore, dropout at the input layer should be avoided.

Dropout at the input layer is better avoided.

These inferences drawn from Equation 4.8 are pertinent to dropout's practical use. Although the equation is derived with linearity assumptions, the results apply to nonlinear conditions.

4.5.3 Dropout Layer

Dropout is implemented in `layers` class in TensorFlow. The primary argument in a `Dropout` layer is the `rate`. Rate is a float between 0 and 1 that defines the fraction of input units to drop (p in Equation 4.8).

As shown in Listing 4.14, dropout is added as a layer after a hidden layer. A dropout layer does not have any trainable parameter (also seen in the model summary in Figure 4.7).

Listing 4.14. MLP with dropout.

```
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation='relu',
  name='hidden_layer_1'))
model.add(Dropout(0.5))
model.add(Dense(16, activation='relu',
  name='hidden_layer_2'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid',
  name='output_layer'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden_layer_1 (Dense)       (None, 32)                2240
_________________________________________________________________
dropout (Dropout)            (None, 32)                0
_________________________________________________________________
hidden_layer_2 (Dense)       (None, 16)                528
_________________________________________________________________
dropout_1 (Dropout)          (None, 16)                0
_________________________________________________________________
output_layer (Dense)         (None, 1)                 17
=================================================================
Total params: 2,785
Trainable params: 2,785
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.7. *MLP with Dropout Model Summary.*

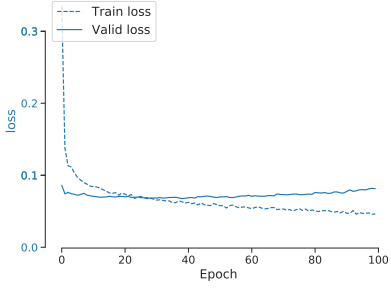
```
13
14 model.compile(optimizer='adam',
15                loss='binary_crossentropy',
16                metrics=['accuracy',
17                         tf.keras.metrics.Recall(),
18                         performancemetrics.F1Score(),
19                         performancemetrics.
                               FalsePositiveRate()]
20             )
```

The results of the model are shown in Figure 4.8a-4.8c. Dropout did improve validation loss. Unlike the previous model, the validation loss is virtually non-increasing.

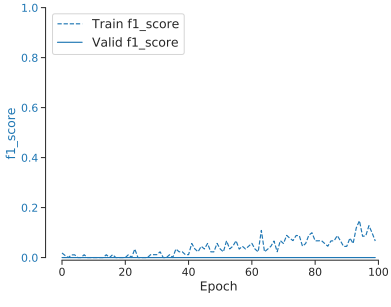
While dropout addressed the overfitting issue, it made the model non-predictive. This is shown in Figure 4.8b and 4.8c where the f1-score and recall are nearly zero.

Dropout sometimes causes this. This phenomenon is typical because a sub-model is learned at a time and the sub-model may not be sufficient to make an accurate prediction.

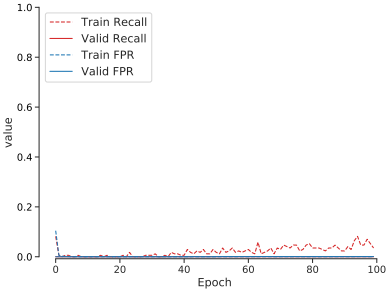
A common resolution to this is increasing the network size: increase the layers and/or their sizes. Moreover, a dropout layer has another argument `noise_shape` to add noise to the inputs. Adding noise can make the model more robust and, therefore, improve accuracy.



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 4.8. *MLP with Dropout Model Results.*

4.6 Class Weights

A rare event problem has very few positively labeled samples. Due to this, even if the classifier is misclassifying the positive labels, their effect on the loss function is minuscule.

Remember the loss function in Equation 4.7, it gives equal importance (weights) to the positive and negative samples. To overcome this, we can overweight the positives and underweight the negative samples. A binary cross-entropy loss function will then be,

$$\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\Big(w_1 y_i \log(p_i) + w_0(1-y_i)\log(1-p_i)\Big) \tag{4.9}$$

where $w_1 > w_0$.

The class-weighting approach works as follows,

- The model estimation objective is to minimize the loss. In a perfect case, if the model could predict all the labels perfectly, i.e. $p_i = 1|y_i = 1$ **and** $p_i = 0|y_i = 0$, the loss will be **zero**. Therefore, the best model estimate is the one with the loss closest to zero.

- With the class weights, $w_1 > w_0$, if the model misclassifies the positive samples, i.e. $p_i \to 0|y_i = 1$, the loss goes **farther away from zero** as compared to if the negative samples are misclassified. In other words, the model training penalizes misclassification of positives more than negatives.

- Therefore, the model estimation strives to correctly classify the minority positive samples.

In principle, any arbitrary weights such that $w_1 > w_0$ can be used. But a rule-of-thumb is,

- w_1, positive class weight = number of negative samples / total samples.

- w_0, negative class weight = number of positive samples / total samples.

Using this thumb-rule, the class weights are defined in Listing 4.15. The computed weights are, `{0: 0.0132, 1: 0.9868}`.

Listing 4.15. Defining Class Weights.

```
class_weight = {0: sum(y_train == 1)/len(y_train),
                1: sum(y_train == 0)/len(y_train)}
```

The model with class weights is shown in Listing 4.16. Except the class weights argument in `model.fit`, the remaining is the same as the baseline model in § 4.4.

Listing 4.16. MLP Model with Class Weights.

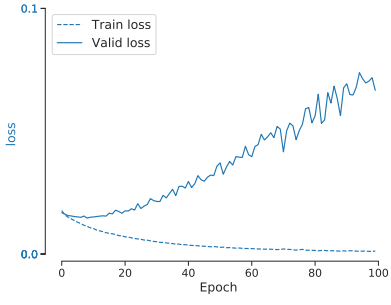
```
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation='relu',
  name='hidden_layer_1'))
model.add(Dense(16, activation='relu',
```

```
6     name='hidden_layer_2'))
7 model.add(Dense(1, activation='sigmoid',
8     name='output_layer'))
9
10 model.summary()
11
12 model.compile(optimizer='adam',
13               loss='binary_crossentropy',
14               metrics=['accuracy',
15                        tf.keras.metrics.Recall(),
16                        performancemetrics.F1Score(),
17                        performancemetrics.
                              FalsePositiveRate()]
18              )
19
20 history = model.fit(x=X_train_scaled,
21                      y=y_train,
22                      batch_size=128,
23                      epochs=100,
24                      validation_data=(X_valid_scaled,
25                        y_valid),
26                      class_weight=class_weight,
27                      verbose=0).history
```

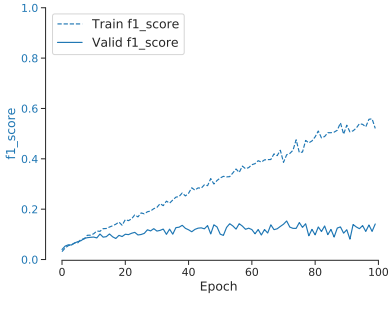
The results of the model with class weights are in Figure 4.9a-4.9c. While the training loss is well-behaved, the validation loss is going upwards. But here it is not necessarily due to overfitting.

It is usual to see such behavior upon manipulating the class weights. Here the validation recall (true positives) is high at the beginning and then decreases along with the false positive rate (false positives). But because the weights of the positive class are higher when both recall and fpr decrease, the validation loss increases faster (effect of lessening true positives are higher) than the reduction (effect of lessening false positives).

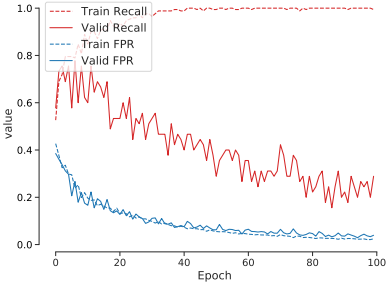
Despite the awkward behavior of the loss, the recall improved significantly. But at the same time, the false positive rate rose to around 4% which is more than desired. This performance can be adjusted by changing the weights.



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 4.9. *MLP with Class Weights Model Results.*

4.7 Activation

Activation functions are one of the primary drivers of neural networks. An activation introduces the non-linear properties to a network.

It is shown in Appendix A that a network with *linear* activation is equivalent to a simple regression model. It is the non-linearity of the activations that make a neural network capable of learning non-linear patterns in complex problems.

But there are a variety of activations, e.g., `tanh`, `elu`, `relu`, etc. Does choosing one over the other improve a model?

Yes. If appropriately chosen, an activation can significantly improve a model. Then, how to choose an appropriate activation?

An appropriate activation is the one that does not have **vanishing**