and/or **exploding** gradient issues. These issues are elaborated next.

4.7.1 What is Vanishing and Exploding Gradients?

Deep learning networks are learned with backpropagation. Backpropagation methods are gradient-based. A gradient-based parameter learning can be generalized as

$$\theta_{n+1} \leftarrow \theta_n - \eta \nabla_\theta \tag{4.10}$$

where n is a learning iteration, η is a learning rate, and ∇_θ is the gradient of the loss $\mathcal{L}(\theta)$ with respect to the model parameters θ.

The equation shows that gradient-based learning iteratively estimates θ. In each iteration, the parameter θ is moved "closer" to its optimal value θ^*.

However, whether the gradient will truly bring θ closer to θ^* will depend on the gradient itself. This is visually illustrated in Figure 4.10a-4.10c.

The gradient, ∇, guides the parameter θ to its optimal value. An apt gradient is, therefore, critical for the parameter's journey to the optimal.

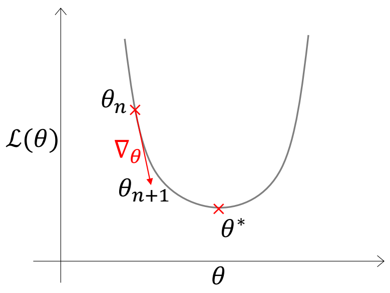
In these figures, the horizontal axis is the model parameter θ, the vertical axis is the loss $\mathcal{L}(\theta)$ and θ^* indicates the optimal parameter at the lowest point of loss.

A stable gradient is shown in Figure 4.10a. The gradient has a magnitude that brings θ_n closer to θ^*.

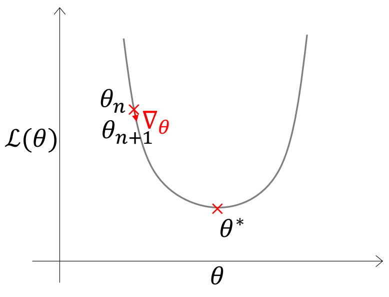
But if the gradient is too small the θ update is negligible. The updated parameter θ_{n+1}, therefore, stays far from θ^*.

The gradient when too small is called the vanished gradient. And this phenomenon is referred to as a **vanishing gradient** issue.

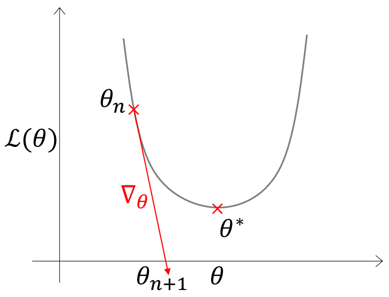
On the other extreme, sometimes the gradient is massive. Depicted in Figure 4.10c, a large gradient moves θ farther away from θ^*.



(a) *Stable gradient.*



(b) *Vanished gradient.*



(c) *Exploded gradient.*

Figure 4.10. *Stable gradient vs. vanishing and exploding gradient.*

This is the **exploding gradient** phenomenon. A large gradient is referred to as an exploded gradient and it makes reaching θ^* rather elusive.

4.7.2 Cause Behind Vanishing and Exploding Gradients

This is explained with the help of expressions in Figure 4.11. The figure is showing the gradient expressions for the layers of the network constructed in this chapter. The expressions are derived in Appendix D.

From the expressions in the figure, the gradients are,

- **Chain multiplied**. The gradients are computed using the chain rule[12]. That is, the partial gradients are successively multiplied.

- **Of the activations**. The gradient ∇_θ is the chain multiplication of the partial gradients of the (layer) activations. In the figure, the $\sigma(\cdot)$ and $g(\cdot)$ are the activation at the output and hidden layers, respectively.
- **Dependent on the feature map**. Each part of the chain multiplication has the layer's input features. The first layer has the input features $\boldsymbol{x}$ and the subsequent layers have the feature maps $\boldsymbol{z}$.

Due to these characteristics, the gradients can,

- **Vanish**. A layer's gradient is a chain of partial gradients multiplied together. At each network depth, another gradient term is included in the chain.

  As a result, the deeper a network, the longer the chain. These gradients are of the activations. If the *activation gradients* are smaller than 1, the overall gradient rapidly gets close to zero. This causes the gradient vanishment issue.
- **Explode**. If the *activation gradients'* or the feature maps' magnitudes are larger than 1, the gradient quickly inflates due to the chain multiplication. Excessive inflation leads to what is termed as gradient explosion.

The vanishing and exploding gradients issues are a consequence of the activation gradient and feature map. The feature map effect is addressed using batch normalization in Ioffe and Szegedy 2015. More work has been done on activations to address these issues and is discussed next.

4.7.3 Gradients and Story of Activations

The vanishing and exploding gradient issues were becoming a bottleneck in developing complex and large neural networks. They were

[12]Chain rule: $f'(x) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial v}\cdots\frac{\partial z}{\partial x}$.

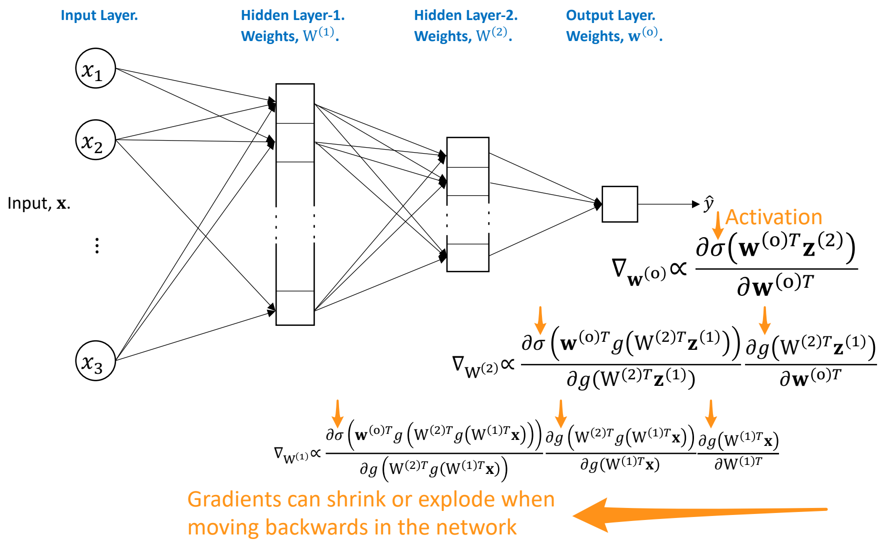

Input Layer.
Hidden Layer-1. Weights, $W^{(1)}$.
Hidden Layer-2. Weights, $W^{(2)}$.
Output Layer. Weights, $\mathbf{w}^{(o)}$.
x_1
x_2
x_3
Input, $\mathbf{x}$.
$\hat{y}$
Activation
$\nabla_{\mathbf{w}^{(o)}} \propto \frac{\partial \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)})}{\partial \mathbf{w}^{(o)T}}$
$\nabla_{W^{(2)}} \propto \frac{\partial \sigma\left(\mathbf{w}^{(o)T} g(W^{(2)T}\mathbf{z}^{(1)})\right)}{\partial g(W^{(2)T}\mathbf{z}^{(1)})} \frac{\partial g(W^{(2)T}\mathbf{z}^{(1)})}{\partial \mathbf{w}^{(o)T}}$
$\nabla_{W^{(1)}} \propto \frac{\partial \sigma\left(\mathbf{w}^{(o)T} g\left(W^{(2)T} g(W^{(1)T}\mathbf{x})\right)\right)}{\partial g\left(W^{(2)T} g(W^{(1)T}\mathbf{x})\right)} \frac{\partial g\left(W^{(2)T} g(W^{(1)T}\mathbf{x})\right)}{\partial g(W^{(1)T}\mathbf{x})} \frac{\partial g(W^{(1)T}\mathbf{x})}{\partial W^{(1)T}}$
Gradients can shrink or explode when moving backwards in the network

Figure 4.11. *Illustration to explain vanishing and exploding gradients. Model training uses the backpropagation technique, which indeed relies on gradients. Gradients are a product of partial derivatives. Each layer adds another partial derivative. Therefore, as backpropagation runs down the lower layers the partial derivatives get stacked up. Depending on their small or large magnitudes, the overall gradient can vanish or explode.*

first resolved to some extent with the rectified linear unit (`relu`) and `leaky-relu` in Maas, Hannun, and Ng 2013.

`Relu` activation is defined as,

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.11}$$

The gradient of `relu` is 1 if $x > 0$ and 0 otherwise. Therefore, when a `relu` unit is "activated," i.e., the input in it is anything greater than zero, its derivative is 1. Due to this, the **gradients vanishment do not happen** for the positive-valued units in an arbitrarily deep network. These units are sometimes also called *active* units.

Additionally, `relu` is nonlinear at $x = 0$ with a saturation region for $x < 0$. A saturation region is where the gradient is zero. The saturation region dampens the activation variance if it is too large in the lower layers. This helps in learning lower level features, e.g., more abstract distinguishing patterns in a classifier.

However, only one saturation region, i.e., zero gradient in one region of x, is desirable. More than one saturation region, like in `tanh` activation (has two saturation region shown in Figure 4.12b), make the variance too small causing gradients vanishment.

A saturation region is the part of a function where the gradient becomes zero. Only one saturation region is desirable.

Theoretically, `relu` resolved the vanishing gradient issue. Still, researchers were skeptical about `relu`. Because without any negative outputs the `relu` activations' means (averages) are difficult to control. Their means can easily stray to large values.

Additionally, `relu`'s gradient is zero whenever the unit is not active. This was believed to be restrictive because the gradient-based backpropagation does not adjust the weights of units that never activate initially. Eventually causing cases where a unit never activates.

To alleviate this, the `relu` authors further developed `leaky-relu`. Unlike `relu`, it has a scaled-down output, $0.01x$ when $x < 0$. The activations are visualized for comparison in Figure 4.12a.

As seen in the figure, the `leaky-relu` has a small but non-zero output for $x < 0$. But this yields a non-zero gradient for every input. That is, it has no saturation region. This did not work in favor of `leaky-relu`.

To resolve the issues in `relu` and `leaky-relu`, `elu` was developed in Clevert, Unterthiner, and Hochreiter 2015. The `elu` activation is defined as,

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(\exp x - 1), & \text{otherwise.} \end{cases} \tag{4.12}$$

`Elu`'s gradient is visualized in Figure 4.12b. In contrast to the `leaky-relu`, `elu` has a saturation in the negative region. As mentioned before, the saturation results in small derivatives which decrease the variance and, therefore, the information is well-propagated to the next layer.

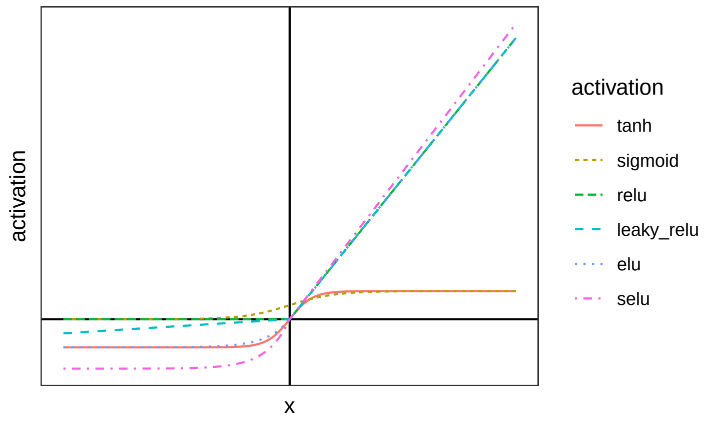
With this property and the non-zero negative outputs, `elu` could enable faster learning as they bring the gradient closer to the natural gradient (shown in Clevert, Unterthiner, and Hochreiter 2015).

However, despite the claims by `elu` or `leaky-relu`, they did not become as popular as `relu`. `Relu`'s robustness made it a default activation for most models.

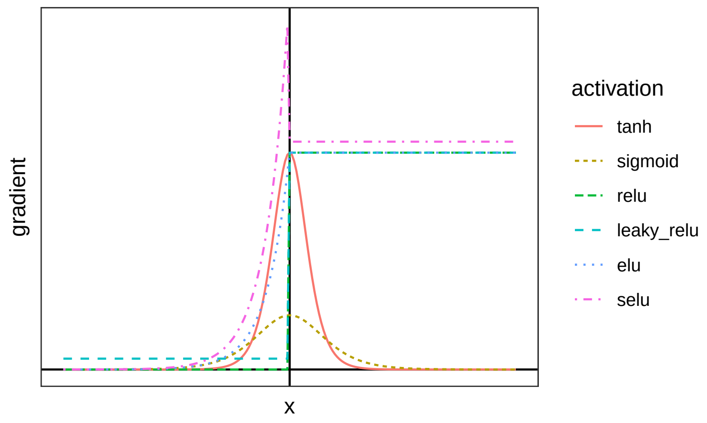
In any case, `relu` and other activations developed thus far could not address the gradient explosion issue. Batch normalization, a computation outside of activation done in a `BatchNormalization` layer, was typically used to address it. Until Scaled Exponential Linear Unit (`selu`) was developed in Klambauer et al. 2017.

`Selu` can construct a self-normalizing neural network. This addresses both the vanishing and exploding gradient issues at the same time.

A `selu` activation, shown in Equation 4.13 below, appears to be a



(a) *Activation output.*



(b) *Activation gradient.*

Figure 4.12. *Activations comparison. The top chart compares the shape of activations $g(x)$ and the bottom compares their gradients $\frac{\partial g(x)}{\partial x}$. An ideal activation for most hidden layers has, 1. nonlinearity which makes a network nonlinear to solve complex problems, 2. a region where the gradient is ≥ 1 and $< 1 + \delta$, where δ is small, to avoid gradient vanishment and explosion, respectively, and 3. a saturation region where the gradient becomes 0 to reduce variance.*

minor change in `elu` in Equation 4.12 with a λ factor.

$$g(x) = \begin{cases} \lambda x, & \text{if } x > 0 \\ \lambda\alpha(\exp x - 1), & \text{otherwise} \end{cases} \tag{4.13}$$

where, $\lambda > 1$.

But Klambauer et al. 2017 proved that the simple change brought an important property of *self-normalization* that none of the predecessors had.

4.7.4 Self-normalization

The activations of a neural network are considered self-normalized if their means and variances across the samples are within predefined intervals. With `selu`, if the input mean and variance are within some intervals, then the outputs' mean and variance will also be in the same respective intervals. That is, the normalization is transitive across layers.

This means that `selu` successively normalizes the layer outputs when propagating through network layers. Therefore, if the input features are normalized the output of each layer in the network will be automatically normalized. And, this normalization **resolves the gradient explosion issue**.

To achieve this, `selu` initializes normalized weight vectors such that $\sum w_i = 0$ and $\sum w_i^2 = 1$ for every layer. The weights are randomly drawn from a truncated normal distribution. For this initialization the best values for the parameters λ and α in Equation 4.13 are derived as 1.0507 and 1.6733 in Klambauer et al. 2017.

The development of `selu` in Klambauer et al. 2017 outlines the generally desired properties of an activation. Among them, the presence of negative and positive values, and a (one) saturation region are the first candidates.

Figure 4.12a and 4.12b display the presence/absence of these properties among the popular activations. Only `elu` and `selu` have both the properties. However, `selu` went beyond `elu` with two additional attributes,

- **Larger gradient**. A gradient larger than one. This increases the variance if it is too small in the lower layers. This would make learning low-level features in deeper networks possible.

  Moreover, the gradient is larger around $x = 0$ compared to `elu` (see Figure 4.12b). This reduces the noise from weaker nodes and guides them to their optimal values faster.

- **Balanced variance**. A fixed point where the variance damping (due to the gradient saturation) is equalized by variance inflation (due to greater than 1 gradient). This controls the activations from vanishing or exploding.

`Selu` properties are irresistible. It performs better in some data sets and especially in very deep networks. But in shallow networks such as the baseline network in this chapter, `selu` might not outperform others.

4.7.5 `Selu` Activation

The MLP model with `selu` activation is shown in Listing 4.17. As shown in the listing, a `selu` activated model requires,

- `kernel_initializer='lecun_normal'`[13]. This initializes the weights by sampling the weight vectors from a truncated normal distribution with mean 0 and standard deviation as

$$\frac{1}{\sqrt{m_{l-1}}}$$

  where m_{l-1} is the number of input units (size of the previous layer).

- `AlphaDropout()`[14]. Standard dropout randomly sets inputs to zero. This does not work with `selu` as it disturbs the activations'

[13]`https://www.tensorflow.org/api_docs/python/tf/initializers/lecun_normal`

[14]`https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/AlphaDropout`

mean and variance. Therefore, a new "alpha dropout" was introduced in Klambauer et al. 2017 in which the inputs are randomly set to α'. α' is set so that the activations' original mean and variance are preserved. This ensures the self-normalizing property of `selu`.

Empirically, an alpha dropout rate of 0.05 or 0.1 leads to a good performance.

Listing 4.17. MLP with `selu` Activation.

```
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation='selu',
  kernel_initializer='lecun_normal'))
model.add(AlphaDropout(0.1))
model.add(Dense(16, activation='selu',
  kernel_initializer='lecun_normal'))
model.add(AlphaDropout(0.1))
model.add(Dense(1, activation='sigmoid'))

model.summary()

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

history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                      y_valid),
                    verbose=0).history
```

4.8 Novel Ideas Implementation

4.8.1 Activation Customization

Activations is an active research field in Deep Learning and still at its nascent stage. It is common among researchers to attempt novel activation ideas. To enable this, custom activation implementation is shown here.

Activations can be defined as a conventional python function. For such definitions, their gradient should also be defined and registered to TensorFlow[15].

However, the gradient definition is usually not required if the activation is defined using TensorFlow functions. TensorFlow has derivatives predefined for its in-built functions. Therefore, explicit gradient declaration is not required. This approach is, therefore, simpler and is practically applicable in most activation definitions.

Here a custom activation, *Thresholded Exponential Linear Unit* (`telu`), is defined in Equation 4.14.

$$g(x) = \begin{cases} \lambda x, & \text{if } x > \tau \\ 0, & \text{if } -\tau \leq x \leq \tau \\ \lambda\alpha(\exp x - 1), & \text{if } x < -\tau \end{cases} \tag{4.14}$$

With this activation, weak nodes smaller than τ will be deactivated. The idea behind thresholding small activations is applying regularization directly through the `telu` activation function.

This custom activation is a modification of `selu`. The activation is defined in Listing 4.18. The first input argument `x` to activation is a tensor. Any subsequent argument is the hyperparameters, that can be defined and set as required. Here, the `threshold`, τ, is the hyperparameter.

Listing 4.18. Custom Activation `telu` definition.

[15] See `tf.RegisterGradient` at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/RegisterGradient`

```
1  from tensorflow.keras import backend as K
2  def telu(x, threshold=0.1):
3      '''
4      Thresholded Exponential linear unit.
5  
6        Arguments:
7        x: Input tensor.
8        alpha: A scalar = 1.6732632, slope
9               of negative section.
10       scale: A scalar = 1.05070098, to
11              keep the gradient > 1 for
12              x > 0.
13 
14       Returns:
15       The thresholded exponential linear
16       activation:
17       scale * x, if x > threshold,
18       0, if -threshold < x < threshold
19       scale * alpha * (exp(x)-1), if
20         ‘x < -threshold‘.
21     '''
22     x_ = tf.keras.activations.selu(x)
23 
24     # Create a tensor of same shape as x
25     # with the threshold in each cell.
26     threshold_ = tf.math.scalar_mul(threshold,
27       K.ones_like(x_))
28 
29     # Creates an identity tensor which
30     # is one if the abs(x) > threshold.
31     threshold_multiplier =
32       K.cast(tf.keras.backend.less(threshold_,
33         K.abs(x_)),
34         dtype='float32')
35 
36     return tf.math.multiply(x_,
37       threshold_multiplier)
```

The activation definition in Listing 4.18 is using in-built TensorFlow functions for every operation. The input `x` is, first, passed through `selu(x)` in line 22 and then thresholded. The thresholding is done by,

- defining a tensor of the same shape as `x` which has a value `1`, if the corresponding absolute of `x` element is greater than the `threshold` and `0`, otherwise.

- Followed by an element-wise multiplication between the vectors using `tf.math.multiply` in line 36.

The custom activation must be tested by defining TensorFlow tensors and passing them through the function. The testing is shown in Listing 4.19.

Listing 4.19. Testing the custom activation `telu`.

```
# Testing TELU
test_x = tf.convert_to_tensor([-1., 1.1, 0.01],
  dtype=tf.float32)

# Sanity test 1: telu output should be
# equal to selu if threshold=0.
tf.print('TELU with threshold=0.0:',
  telu(test_x, threshold=0.))
tf.print('SELU for comparison: ',
  tf.keras.activations.selu(test_x))

# Output:
# TELU with threshold=0.0: [-1.11133075 1.15577114 0.0105070099]
# SELU for comparison:     [-1.11133075 1.15577114 0.0105070099]

# Sanity test 2: telu should make
# activations < threshold.
tf.print('TELU default setting: ',
 telu(test_x))  # default threshold = 0.1

# Output:
# TELU default setting:    [-1.11133075 1.15577114 0]
```

First, a sanity test is done by setting the threshold as 0.0. In this setting, the output should be equal to the output of a `selu` activation. Second, `telu` functionality is validated by setting the threshold as 0.1.

With this threshold, the input smaller than the threshold is made 0. The outputs are shown under comments in the listing.

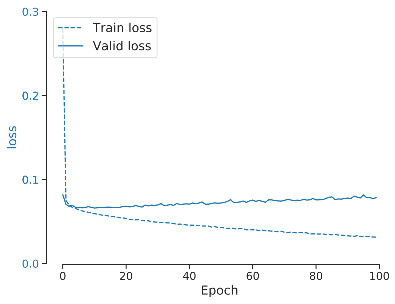
The activation definition passed both tests. In general, it is important to run various tests on custom functions before moving forward. Because the farther we go with custom functions, the harder it becomes to pinpoint the problem if any arises.

In Listing 4.20, an MLP model is trained with the custom `telu` activation.

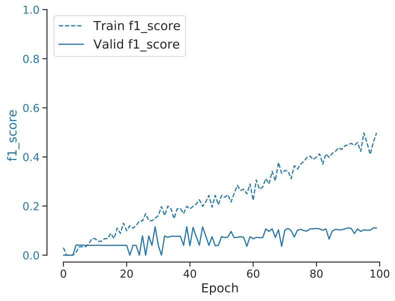
Listing 4.20. MLP model with the custom `telu` activation.

```
model = Sequential()
model.add(Input(shape=(N_FEATURES, )))
model.add(Dense(32, activation=telu))
model.add(Dense(16, activation=telu))
model.add(Dense(1, activation='sigmoid'))

model.summary()

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

history = model.fit(x=X_train_scaled,
                    y=y_train,
                    batch_size=128,
                    epochs=100,
                    validation_data=(X_valid_scaled,
                      y_valid),
                    verbose=0).history
```

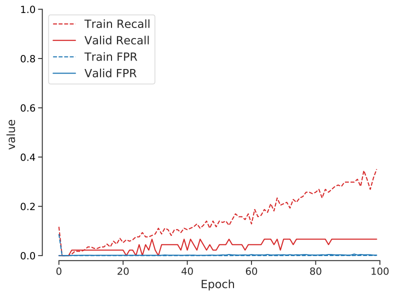
`Telu` activation performed at par with the baseline. While the baseline model had increasing validation, `telu` resolved the increasing validation loss issue without sacrificing the accuracy.



(a) *Loss.*



(b) *F1-score.*



(c) *Recall and FPR.*

Figure 4.13. *MLP with* **`telu`** *activation results.*

The development and results from a custom **`telu`** *activation here demonstrates that there is significant room for research in activation.*

4.8.2 Metrics Customization

Looking at suitably chosen metrics for a problem tremendously increases the ability to develop better models. Although a metric does not directly improve model training but it helps in a better model selection.

Several metrics are available outside TensorFlow such as in `sklearn`. However, they cannot be used directly during model training in TensorFlow. This is because the metrics are computed while processing batches during each training epoch.

Fortunately, TensorFlow provides the ability for this customization. The custom-defined metrics `F1Score` and `FalsePositiveRate` are provided in the user-defined `performancemetrics` library. Learning the programmatic context for the customization is important and, therefore, is elucidated here.

The TensorFlow official guide shows the steps for writing a custom metric[16]. It instructs to create a new subclass inheriting the `Metric` class and work on the following definitions for the customization,

- `__init__()`: All the state variables should be created in this method by calling `self.add_weight()` like:
  `self.var = self.add_weight(...)`.
- `update_state()`: All updates to the state variables should be done as `self.var.assign_add(...)`.
- `result()`: The final result from the state variables is computed and returned in this function.

Using these instructions, the `FalsePositiveRate()` custom metric is defined in Listing 4.21. Note that `FalsePositives` metric is already present in TensorFlow. But drawing the false positive rate (a ratio) from it during training is not direct and, therefore, the `FalsePositiveRate()` is defined.

Listing 4.21. Custom FalsePositiveRate Metric Definition.

```python
class FalsePositiveRate(tf.keras.metrics.Metric):
    def __init__(self, name='false_positive_rate',
       **kwargs):
        super(FalsePositiveRate, self).
        __init__(name=name, **kwargs)
        self.negatives = self.add_weight(name='
            negatives',
          initializer='zeros')
        self.false_positives = self.add_weight(
          name='false_negatives',
          initializer='zeros')
```

[16]tensorflow/python/keras/metrics.py at `shorturl.at/iqDS7`

```

    def update_state(self,
      y_true, y_pred, sample_weight=None):
        '''
        Arguments:
        y_true   The actual y. Passed by
                 default to Metric classes.
        y_pred   The predicted y. Passed
                 by default to Metric classes.

        '''
        # Compute the number of negatives.
        y_true = tf.cast(y_true, tf.bool)

        negatives = tf.reduce_sum(tf.cast(
          tf.equal(y_true, False), self.dtype))

        self.negatives.assign_add(negatives)

        # Compute the number of false positives.
        y_pred = tf.greater_equal(
            y_pred, 0.5
        )  # Using default threshold of 0.5 to
           # call a prediction as positive labeled.

        false_positive_vector =
          tf.logical_and(tf.equal(y_true, False),
            tf.equal(y_pred, True))
        false_positive_vector = tf.cast(
           false_positive_vector,
          self.dtype)
        if sample_weight is not None:
            sample_weight = tf.cast(sample_weight,
              self.dtype)
            sample_weight = tf.broadcast_weights(
              sample_weight, values)
            values = tf.multiply(
               false_positive_vector,
              sample_weight)

        false_positives = tf.reduce_sum(
           false_positive_vector)
```

```

            self.false_positives.assign_add(
                false_positives)

    def result(self):
        return tf.divide(self.false_positives,
          self.negatives)
```

How do metrics computation and customization work?

A model training is done iteratively. The iterations happen at multiple levels. The iteration levels for multi-layer perceptron training are laid out in Figure 4.14.

The topmost iteration level is *epochs*. Within an epoch, a model is trained iteratively over randomly selected batches. The batches are the second level of iteration.

Multiple samples are present within a batch. The model can be trained by processing one sample at a time. But they are processed together as a batch (as shown in Equation 4.4) for computational efficiency. The `sample` is, therefore, grayed in the figure indicating it a logical iteration instead of an actual one.

During batch processing, all the model parameters—weights and biases—are updated. Simultaneously, the *states* of a metric are also updated. Upon processing all the batches in an epoch, both the estimated parameters and computed metrics are returned. Note that all these operations are enclosed within an epoch and no values are communicated between two epochs.

The meaning of the metrics state, metrics computation, and the programmatic logic in Listing 4.21 for defining `FalsePositiveRate` are enumerated below.

- The class `FalsePositiveRate()` is inheriting the `Metric` class in TensorFlow. As a result, it automatically has the `__init__()`, `update_state()`, and `result()` definitions. These definitions will be overwritten during the customization.

- `__init__()` is the entry gate to a metric class. It initializes the

```
epoch
└─ batch
   └─ sample
```

Figure 4.14. *Levels of iteration in MLP training. The iterations are interpreted up to the sample level. However, computationally all samples in a batch are processed together using tensor operations. Hence, the `sample` is grayed here to indicate it is only a logical iteration level.*

state variables. **State variables** are used for metric computation.

- The false-positive rate is the ratio of false positives over the negatives. Therefore, `false_positives` and `negatives` become the state variables.
- The state variables are prefixed with `self.`. A `self` variable can be accessed and updated from any definition in the class.
- The states—false positives and negatives—are initialized to zero in `__init__()`. That is, at the onset of an epoch all the states are reset to their initial values. In some problems, the initial value can be other than zero.
- After `__init__()`, the program enters `update_state()`. This function can be imagined as the metric processing unit. It is run for every batch in an epoch. Inside this function, the metric states are updated from the outcome derived from a batch.
- `update_state()` has the actual labels `y_true` and the predictions `y_pred` for the samples in the batch as default arguments.
- Line 23-28 in the function are computing and updating the `self.negatives` state variable. The original labels `y_true` in the inputted data are numeric 0 and 1. They are first converted into boolean in Line 23. The `negatives` in the batch are then computed by finding all the `False` (the negative samples) in the boolean `y_true` vector followed by converting it back to numeric 0/1 and summing the now numeric vector.

- The `negatives` computed in Line 25 is the number of negative samples in the batch under process. `negatives` is a local variable in its enclosing function `update_state()`. It is added to the metric state variable `self.negatives` (a class variable) in Line 28.
- Similarly, the false positives in the batch is computed in Line 31-51. False positives is derived by comparing the predictions `y_pred` with the actuals `y_true`.
- To get to the false positives, the first step is to convert `y_pred` to boolean. `y_pred` is the output of a *sigmoid* activated output layer. Thus, it is a continuous number in $(0, 1)$.
- `y_pred` is converted to boolean using a default threshold of 0.5[17]. The predictions in `y_pred` greater than the threshold are made `True` and `False`, otherwise.
- A false positive is when the actual label `y_true` is `False` but the prediction `y_pred` is `True`. That is, the model incorrectly predicted a negative sample as positive. This logical comparison is done in Line 36.
- The output of the logical comparison is a boolean vector `false_positive_vector` which is converted to numeric 0/1 in Line 39.
- The `false_positives` in the batch is then computed by summing the vector in Line 49.
- The possibility of a not `None` sample weight is accounted for in Line 26-29.
- Same as before, the `self.false_positives` state is then updated in Line 51.
- After the epoch has iterated through all the batches the metric state variables `self.negatives` and `self.false_positives` will have stored the totals for the entire data set.

[17]This is a default value used in practice. If needed, it can be changed to any other desired value.

- The state `self.negatives` has the total negative samples in the data set. This is a constant and, therefore, `self.negatives` will remain the same in all epochs. `self.false_positives`, on the other hand, is a result of the goodness of model fitting. And so it changes (ideally reduces) over successive epochs.

- Finally, the program goes to `result()` and yields the metric's value. This is the "exit" gate of the metric class. Here the false positive rate is computed by dividing the metric states `self.false_positives` and `self.negatives`.

- **Important**: a metric state should be additive only. This is because the `update_state()` can only increment or decrement the state variable. For instance, if the false-positive rate—a ratio and, thus, non-additive—was created as the state variable and updated in `update_state()` it would yield $\sum_{i \in \texttt{batches}} \frac{\texttt{false positives}_i}{\texttt{negatives}_i}$ instead of the desired $\frac{\sum_{i \in \texttt{batches}} \texttt{false positives}_i}{\sum_{i \in \texttt{batches}} \texttt{negatives}_i}$. In general, any non-additive computation should, therefore, be done in `result()`.

The state in metric customization should be additive only.

Similarly, the `F1Score` custom metric is defined in Listing 4.22.

Listing 4.22. Custom F1Score Metric Definition.

```python
class F1Score(tf.keras.metrics.Metric):
    def __init__(self, name='f1_score', **kwargs):
        super(F1Score, self).__init__(name=name, **
            kwargs)
        self.actual_positives = \
          self.add_weight(name='actual_positives',
                          initializer='zeros')
        self.predicted_positives =
          self.add_weight(name='predicted_positives',
                          initializer='zeros')
        self.true_positives =
          self.add_weight(name='true_positives',
                          initializer='zeros')

    def update_state(self,
      y_true, y_pred, sample_weight=None):
        '''
        Arguments:
        y_true    The actual y. Passed by default
                  to Metric classes.
        y_pred    The predicted y. Passed by
                  default to Metric classes.

        '''
        # Compute the number of negatives.
        y_true = tf.cast(y_true, tf.bool)

        actual_positives = tf.reduce_sum(
            tf.cast(tf.equal(y_true, True), self.dtype))
        self.actual_positives.assign_add(actual_positives)

        # Compute the number of false positives.
        y_pred = tf.greater_equal(
            y_pred, 0.5
        )  # Using default threshold of 0.5 to call a prediction as positive labeled.

        predicted_positives = tf.reduce_sum(
            tf.cast(tf.equal(y_pred, True),
                    self.dtype))
        self.predicted_positives.assign_add(predicted_positives)

        true_positive_values =
          tf.logical_and(tf.equal(y_true, True),
                         tf.equal(y_pred, True))
        true_positive_values =
          tf.cast(true_positive_values, self.dtype)

        if sample_weight is not None:
            sample_weight =
              tf.cast(sample_weight, self.dtype)
            sample_weight =
              tf.broadcast_weights(sample_weight, values)
            values =
              tf.multiply(true_positive_values, sample_weight)

        true_positives = tf.reduce_sum(true_positive_values)

        self.true_positives.assign_add(true_positives)

    def result(self):
        recall =
          tf.math.divide_no_nan(
            self.true_positives,
            self.actual_positives)
        precision =
          tf.math.divide_no_nan(
            self.true_positives,
            self.predicted_positives)
        f1_score =
          2 * tf.math.divide_no_nan(
            tf.multiply(recall, precision),
            tf.math.add(recall, precision))

        return f1_score
```

4.9 Models Evaluation

Several models were built in this chapter. Out of them, one model should be selected as the *final* model. The performance of the final model is evaluated on the test data. This is a traditional process of model building and selection. For this purpose, the data set was initially

Table 4.1. MLP models comparison. The red highlighted values indicate an undesirable or poor result.

	Validation			
Model	**Loss**	**F1-score**	**Recall**	**FPR**
Baseline	Increasing	0.13	0.08	0.001
Dropout	Non-increasing	0.00	0.00	0.000
Class weights	Increasing	0.12	0.31	0.102
`selu`	Non-increasing	0.04	0.02	0.001
`telu` (custom)	Non-increasing	0.12	0.08	0.001

split in train, valid, and test.

The selection is made by comparing the validation results. These results are summarized in Table 4.1. The baseline model has higher accuracy measures but increasing validation loss indicating potential overfitting. Dropout resolves the overfitting but the accuracy goes to zero. Class weights boosted the accuracy but also has a high false-positive rate. The `selu` activation attempted after that had significantly lower f1-score than the baseline.

The custom activation `telu` performed relatively better than the others. It has non-increasing validation loss and close to the baseline accuracy. Therefore, `telu` activated model is selected as the final model.

This does not mean `telu` will be the best choice in other cases. The process of building multiple models and selection should be followed for every problem.

The final selected model is evaluated using `model.evaluate()` function. The `evaluate` function applies the trained model on test data and returns the performance measures defined in `model.compile()`.

The evaluation and test results are shown in Figure 4.23.

Listing 4.23. MLP final selected model evaluation.

```
# Final model
model.evaluate(
            x=X_test_scaled,
            y=y_test,
            batch_size=128,
             verbose=1)

# loss: 0.0796 - accuracy: 0.9860 -
# recall_5: 0.0755 - f1_score: 0.1231 -
# false_positive_rate: 0.0020
```

Multi-layer perceptrons are elementary deep learning models. Any reasonable result from MLPs act as a preliminary screening that there are some predictive patterns in the data. This lays down a path for further development with different network architectures.

4.10 Rules-of-thumb

This chapter went through a few MLP model constructions. Even in those few models, several modeling constructs and their settings were involved. In practice, there are many more choices. And they could be overwhelming.

Therefore, this chapter concludes with some thumb-rules to make an initial model construction easier.

- **Number of layers**. Start with two hidden layers (this does not include the last layer).
- **Number of nodes (size) of intermediate layers**. A number from a geometric series of 2: 1, 2, 4, 8, 16, 32, ... The first layer should be around half of the number of input data features. The next layer size is half of the previous and so on.
- **Number of nodes (size) of the output layer**.
  - **Classification**. If binary classification then the size is one. For a multi-class classifier, the size is the number of classes.
  - **Regression**. If a single response then the size is one. For multi-response regression, the size is the number of responses.
- **Activation**.
  - **Intermediate layers**. Use `relu` activation.

  - **Output layer**. Use `sigmoid` for binary classification, `softmax` for a multi-class classifier, and `linear` for regression.

- **Dropout**. Do not add dropout in a baseline MLP model. It should be added to a large or complex network. Dropout should certainly be not added to the input layer.

- **Data Preprocessing**. Make the features X as numeric by converting any categorical columns into one-hot-encoding. Then, perform feature scaling. Use `StandardScaler` if the features are unbounded and `MinMaxScaler` if bounded.

- **Split data to train, valid, test**. Use `train_test_split` from `sklearn.model_selection`.

- **Class weights**. Should be used with caution and better avoided for extremely imbalanced data. If used in a binary classifier, the weights should be: 0: number of 1s / data size, 1: number of 0s / data size.

- **Optimizer**. Use `adam` with its default learning rate.

- **Loss**.
  - **Classification**. For binary classification use `binary_crossentropy`. For multiclass, use `categorical_crossentropy` if the labels are one-hot-encoded, otherwise use `sparse_categorical_crossentropy` if the labels are integers.
  - **Regression**. Use `mse`.

- **Metrics**.
  - **Classification**. Use `accuracy` that shows the percent of correct classifications. For imbalanced data, also include `Recall`, `FalsePositiveRate`, and `F1Score`.
  - **Regression**. Use `RootMeanSquaredError()`.

- **Epochs**. Set it as 100 and see if the model training shows decreasing loss and any improvement in the metrics over the epochs.

- **Batch size**. Choose the batch size from the geometric progression of 2. For imbalanced data sets have larger value, like 128, otherwise, start with 16.

For advanced readers,

- **Oscillating loss**. If the oscillating loss is encountered upon training then there is a convergence issue. Try reducing the learning rate and/or change the batch size.

- **Oversampling and undersampling**. Random sampling will work better for multivariate time series than synthesis methods like `SMOTE`.

- **Curve shifting**. If the time-shifted prediction is required, for example, in an early prediction use curve shifting.

- **Selu activation**. `selu` activation has been deemed as better for large networks. If using `selu` activation then use `kernel_initializer='lecun_normal'` and `AlphaDropout()`. In `AlphaDropout`, use the rate as `0.1`.

4.11 Exercises

1. The chapter mentioned few important properties. In their context, show,

   (a) Why a linearly activated MLP model is equivalent to linear regression? Refer to Appendix A.

   (b) Why is a neural network of a single layer, unit sized layer, and `sigmoid` activation the same as logistic regression?

   (c) How the loss function with dropout under linear activation assumption (Equation 4.8) contains an $\mathcal{L}_2$ regularization term? Refer to Baldi and Sadowski 2013.

   Also, show how the dropout approach is similar to an ensemble method? Refer to Srivastava et al. 2014.

2. **Temporal features**. In (multivariate) time series processes, temporal patterns are naturally present. These patterns are expected to be predictive of the response. Temporal features can be added as predictors to learn such patterns.

   Add the following set of features (one set at a time) to the *baseline* and *final* model of your data. Discuss your findings.

   (a) Up to three lag terms for y and each x's. That is, create samples like,

   $$(\text{response: } y_t, \text{predictors: } y_{t-1}, y_{t-2}, y_{t-3}, \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \boldsymbol{x}_{t-3})$$

   (b) Lag terms up to 10. Does increasing the lags have any effect on the model performance?

   (c) The first derivative of x's, i.e., a sample tuple will be,

   $$(\text{response: } y_t, \text{predictors: } \boldsymbol{x}_t, \boldsymbol{x}_t - \boldsymbol{x}_{t-1})$$

   (d) Second derivatives of x's, i.e.

   $$(\text{response: } y_t, \text{predictors: } \boldsymbol{x}_t, (\boldsymbol{x}_t - \boldsymbol{x}_{t-1}) - (\boldsymbol{x}_{t-1} - \boldsymbol{x}_{t-2}))$$

   (e) Both, first and second derivatives together as predictors.

   (f) Does the derivatives add to the predictability? Why?

   (g) What is the limitation in manually adding features?

(h) (Optional) Frequency domain features. For temporal processes, the features in the frequency domain are known to have good predictive signals. They are more robust to noise in the data. Add frequency domain features as predictors. Discuss your findings.

(i) (Optional) The MLP model constructed in the chapter does not have any temporal features. By itself, the MLP cannot learn temporal patterns. Why did the MLP model without temporal features could still work?

3. **Accuracy metric—Diagnostics Odds Ratio.** Imbalanced data problems need to be evaluated with several non-traditional metrics. A diagnostic odds ratio is one of them.

   (a) Explain diagnostic odds ratio and its interpretation in the context of imbalanced class problems.

   (b) Build a custom metric for the diagnostics odds ratio.

   (c) Add it to the *baseline* and *final* models. Discuss the results.

4. **Batch normalization.** It is mentioned in § 4.7 that extreme feature values can cause vanishing and exploding gradient issues. Batch normalization is another approach to address them.

   (a) Explain the batch normalization approach. Describe it alongside explaining Algorithm 1 in Ioffe and Szegedy 2015.

   (b) Add `BatchNormalization` in TensorFlow to the *baseline* and *final* models. Discuss your results.

   (c) Train the models without feature scaling. Can batch normalization work if the input features are not scaled?

   (d) (Optional) Define a custom layer implementing Algorithm 1 in Ioffe and Szegedy 2015[18]

5. **Dropout.** Understand dropout behavior.

   (a) Dropout does not necessarily work well in shallow MLPs. Build an MLP with more layers and add dropout layers.

[18]The noise and gaussian dropout layers definition here, `https://github.com/tensorflow/tensorflow/blob/v2.1.0/tensorflow/python/keras/layers/noise.py` is helpful.

(b) Build a two hidden layer MLP with larger layer sizes along with dropout.

(c) Discuss the results of (a) and (b).

(d) Srivastava et al. 2014 state that a multiplicative Gaussian noise (now known as Gaussian Dropout) can work better than a regular dropout. Explain the reasoning behind this theory. Repeat (a)-(c) with Gaussian dropout and discuss the results.

6. **Activation.**

(a) `Selu` has one of the best properties among the existing activations. It is believed to work better in deeper networks. Create an MLP network deeper than the baseline and use `selu` activation. Discuss the results.

(b) (Optional) *Thresholded exponential linear unit* (`telu`) is a new activation developed in this chapter. It performed better compared to others. This shows that there is room for developing new activations that might outperform the existing ones.

In this spirit, make the following modification in `telu`.

The threshold τ in Equation 4.14 is fixed. Make τ adaptive by making it proportional to the standard deviation of its input x. Change $\tau = 0.1\sigma_x$ in Equation 4.14 and build customized activation.

The idea is to adjust the threshold based on the input variance.

Apply this customized activation on the *baseline* model and discuss the results.

(c) (Optional) Can the results be further improved with the activation customization? Define your custom activation and test it.

Chapter 5

Long Short Term Memory Networks

5.1 Background

"Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence."

– in Olah 2015.

Sequences and time series processes are like essays. The order of words in an essay and, likewise, the observations in sequences and time series processes are important. Due to this, they have temporal patterns. Meaning, the previous observations (the memory) has an effect on the future.

Memory persistence is one approach to learn such temporal patterns. Recurrent neural networks (RNN) like long- and short-term memory networks were conceptualized for this purpose.

RNNs constitute a very powerful class of computational models capable of learning arbitrary dynamics. They work by keeping a memory of patterns in sequential orders. It combines knowledge from the past memories with the current information to make a prediction.

RNN development can be traced back to Rumelhart, G. E. Hinton, and R. J. Williams 1985. Several RNN variants have been developed since then. For example, Elman 1990, Jordan 1990, and time-delay neural networks by Lang, Waibel, and G. E. Hinton 1990.

However, the majority of the RNNs became obsolete because of their inability to learn long-term memories. This was due to the vanishing gradient issue (explained in the RNN context in § 5.2.8).

The issue was addressed with the development of *long short term memory* (LSTM) networks by Hochreiter and Schmidhuber 1997. LSTMs introduced the concept of cell state that holds the long- and short-term memories.

This ability was revolutionary in the field of RNNs. A majority of the success attributed to RNNs comes from LSTMs. They have proven to work significantly better for most of the temporal and sequential data.

At its inception, LSTMs quickly set several records. It outperformed real-time recurrent learning, back-propagation through time, Elman-nets, and others, popular at the time. LSTM could solve complex and long time-lag tasks that were never solved before.

Developments in and applications of LSTMs continued over the years. Today, they are used by Google and Facebook for text translation (Yonghui Wu et al. 2016; Ong 2017). Apple and Amazon use it for speech recognition in Siri (C. Smith 2016) and Alexa (Vogels 2016), respectively.

Despite the successes, LSTMs have also faced criticisms from some researchers. There has been debate around the need for LSTMs after the development of transformers and attention networks (Vaswani et al. 2017).

Regardless of the varied beliefs, LSTMs and other RNNs still stand as major pillars in deep learning. In addition to their superiority in various problems, there is an immense scope of new research and development.

This chapter begins by explaining the fundamentals of LSTM in § 5.2. LSTMs are one of the most complex constructs in deep learning.

This section attempts at deconstructing it and visualizing each element for a clearer understanding.

The subsections in § 5.2 show an LSTM cell structure (§ 5.2.2), the state mechanism that causes memory persistence (§ 5.2.3), the operations behind the mechanism (§ 5.2.4), the model parameters (§ 5.2.6), and the training iterations (§ 5.2.7). Importantly, LSTMs are capable of persisting long-term memories due to a stable gradient. This property is articulated in § 5.2.8.

Moreover, LSTM networks have an intricate information flow which is seldom visualized but paramount to learn. § 5.3 explains LSTM cell operations and information flow in a network with expanded visual illustrations. The subsections, therein, elucidates *stateless* and *stateful* networks (§ 5.3.2), and the importance of (not) returning sequence outputs (§ 5.3.3).

The chapter then exemplifies an end-to-end implementation of a (baseline) LSTM network in § 5.4 and 5.5. Every step from data preparation, *temporalization*, to network construction are shown. A few network improvement techniques such as *unrestricted* network, *recurrent dropout*, *go-backwards*, and *bi-directional* are explained and implemented in § 5.6.1-5.6.5.

LSTMs have a rich history of development. It has matured to the current level after several iterations of research. § 5.7 walks through a brief history of recurrent neural networks and the evolution of LSTMs.

Lastly, the networks are summarized in § 5.8 and a few rules-of-thumb for LSTM networks are given in § 5.9.

5.2 Fundamentals of LSTM

LSTMs are one of the most abstruse theories in elementary deep learning. Comprehending the fundamentals of LSTM from its original paper(s) can be intimidating.

For an easier understanding, it is deconstructed to its elements and every element is explained in this section. This begins with a typical neural network illustration in Figure 5.1.

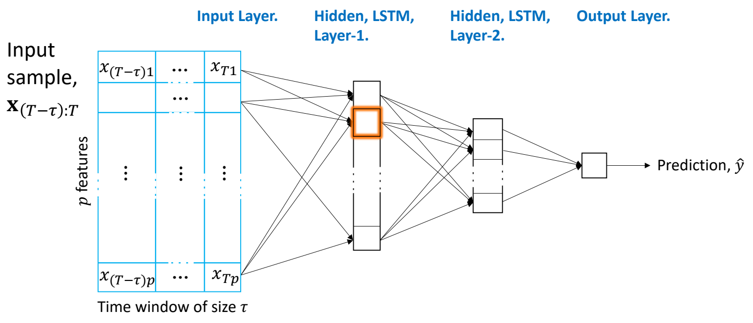


Figure 5.1. *A high-level representation of an LSTM network. The input is a time-window of observations. An LSTM network is designed to learn spatio-temporal relationships from these time-window inputs. The orange highlighted box represents an LSTM cell in the LSTM layer. Each cell learns a distinctive spatio-temporal feature from the inputs.*

5.2.1 Input to LSTM

The input to an LSTM layer is a time-window of observations. In Figure 5.1, it is denoted as $\boldsymbol{x}_{(T-\tau):T}$. This represents p-dimensional observations in a window of size τ.

This window of observations serves as an input sample. The window allows the network to learn the spatial and temporal relationships (remember Figure 2.1c in Chapter 2).

5.2.2 LSTM Cell

The hidden layers in Figure 5.1 are LSTM. The nodes in a layer is an LSTM *cell*—highlighted in orange. A node in LSTM is called a *cell* because it performs a complex biological cell-like multi-step procedure.

This multi-step procedure is enumerated in § 5.2.4. Before getting there, it is important to know the distinguishing property that the cell mechanism brings to LSTM.

The cell mechanism in LSTM has an element called *state*. A cell state can be imagined as a *Pensieve* in *Harry Potter*.

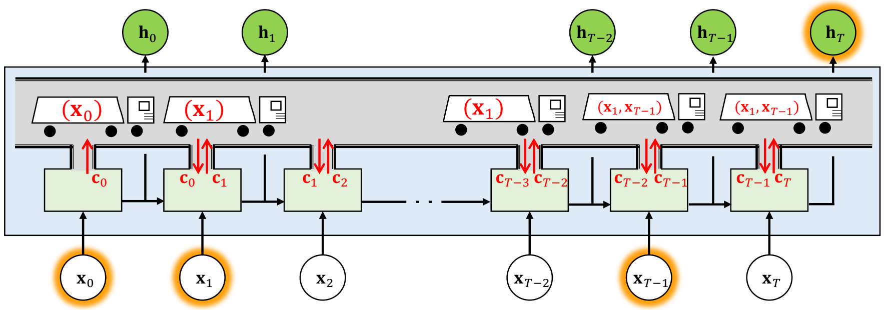
"I use the Pensieve. One simply siphons the excess thoughts from one's mind, pours them into the basin, and examines them at one's leisure. It becomes easier to spot patterns and links; you understand when they are in this form."
–Albus Dumbledore explaining a Pensieve in *Harry Potter and the Goblet of Fire.*

Like a Pensieve, sans magic, a cell state preserves memories from current to distant past. Due to the cell state, it becomes easier to spot patterns and links by having current and distant memories. And, this makes the difference for LSTMs. The state and its mechanism are elaborated in detail next.

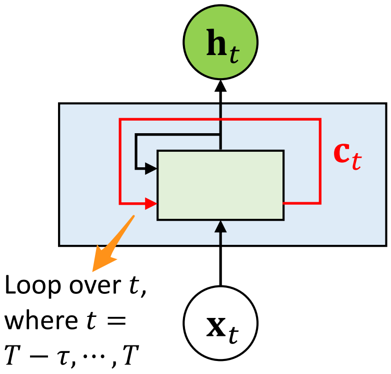
5.2.3 State Mechanism

The cell state mechanism is explained with the help of an intuitive illustration in Figure 5.2a. In the figure, the blue-shaded larger box denotes an LSTM cell. The cell operations are deconstructed inside the box and explained below.

- The input sample to a cell is a time-window of observations $\boldsymbol{x}_{(T-\tau):T}$. For simplicity, $T - \tau$ is replaced with 0 in the figure. The observations are, thus, shown as $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$.
- The cell sequentially processes the time-indexed observations.
- The iterations are shown as green boxes sequentially laid inside the deconstructed cell.
- A green box takes in one time-step $\boldsymbol{x}_t$. It performs some operations to compute the cell state, $\boldsymbol{c}_t$, and the output, $\boldsymbol{h}_t$.
- Like the other RNNs, the hidden output $\boldsymbol{h}_t$ is transmitted to the next iteration and, also, returned as a cell output. This is shown with branched arrows with horizontal and vertical branches carrying $\boldsymbol{h}_t$. The horizontal branch goes to the next green box (iteration) and the vertical branch exits the cell as an output.
- Differently from the other RNNs, an LSTM cell also transmits the cell state $\boldsymbol{c}_t$.


$\mathbf{h}_0$
$\mathbf{h}_1$
$\mathbf{h}_{T-2}$
$\mathbf{h}_{T-1}$
$\mathbf{h}_T$
$(\mathbf{x}_0)$
$(\mathbf{x}_1)$
$(\mathbf{x}_1)$
$(\mathbf{x}_1, \mathbf{x}_{T-1})$
$(\mathbf{x}_1, \mathbf{x}_{T-1})$
$\mathbf{c}_0$
$\mathbf{c}_0$ $\mathbf{c}_1$
$\mathbf{c}_1$ $\mathbf{c}_2$
$\mathbf{c}_{T-3}$ $\mathbf{c}_{T-2}$
$\mathbf{c}_{T-2}$ $\mathbf{c}_{T-1}$
$\mathbf{c}_{T-1}$ $\mathbf{c}_T$
$\mathbf{x}_0$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_{T-2}$
$\mathbf{x}_{T-1}$
$\mathbf{x}_T$

(a) *Illustration of an unwrapped LSTM cell mechanism showing the time-step iterations. The input to a cell is the time-window of observations* $\boldsymbol{x}_{(T-\tau):T}$. *The index* $(T-\tau)$ *is replaced with* 0 *for simplicity. The LSTM cell acts as a memory lane in which the cell states carry the long- and short-term memories through an imaginary truck of information.*



(b) *A condensed form of the LSTM cell mechanism. In actuality, the cell states* $(\boldsymbol{h}_t, \boldsymbol{c}_t)$ *are re-accessed iteratively for* $t = (T-\tau), \ldots, T$. *A succinct representation of this is shown with the looping arrows.*

Figure 5.2. *An unwrapped and condensed (wrapped) illustration of LSTM cell state mechanism.*

- Imagine the iterations along the time-steps, $t = 0, \dots, T$, in a cell as a drive down a lane. Let's call it a "memory lane." A green box (the time-step iteration) is a station on this lane. And, there is a "truck of information" carrying the cell state, i.e., the memory.
- The truck starts from the left at the first station. At this station, the inputted observation $\boldsymbol{x}_0$ is assessed to see whether the information therein is relevant or not. If yes, it is loaded on to the truck. Otherwise, it is ignored.
- The *loading* on the truck is the cell state. In the figure's illustration, $\boldsymbol{x}_0$ is shown as important and loaded to the truck as a part of the cell state.
- The cell state $\boldsymbol{c}_t$ as truckloads are denoted as $(\boldsymbol{x}.)$ to express that the state is some function of the $\boldsymbol{x}$'s and not the original $\boldsymbol{x}$.
- The truck then moves to the next station. Here it is unloaded, i.e., the state/memory learned thus far is taken out. The station assesses the unloaded state alongside the $\boldsymbol{x}$ available in it.
- Suppose this station is t. Two assessments are made here. First, is the information in the $\boldsymbol{x}_t$ at the station relevant? If yes, add it to the state $\boldsymbol{c}_t$.

  Second, in the presence of $\boldsymbol{x}_t$ is the memory from the prior $\boldsymbol{x}$'s still relevant? If irrelevant, forget the memory.

  For example, the station next to $\boldsymbol{x}_0$ is $\boldsymbol{x}_1$. Here $\boldsymbol{x}_1$ is found to be relevant and added in the state. At the same time, it is found that $\boldsymbol{x}_0$ is irrelevant in the presence of $\boldsymbol{x}_1$. And, therefore, the memory of $\boldsymbol{x}_0$ is taken out of the state. Or, in LSTM terminology, $\boldsymbol{x}_0$ is forgotten.
- After the processing, the state is then loaded back on the truck.
- The process of loading and unloading the truck-of-information is repeated till the last $\boldsymbol{x}_T$ in the sample. Further down the lane, it is shown that $\boldsymbol{x}_2, \boldsymbol{x}_{T-2}$ and $\boldsymbol{x}_T$ contain irrelevant information. They are ignored while $\boldsymbol{x}_{T-1}$ was added as its information might be absent in the state $\boldsymbol{c}_{T-2}$ learned before reaching it.

- The addition and omission of the timed observations of a sample makes up the **long-term** memory in $\boldsymbol{c}_t$.

- Moreover, the intermediate outputs $\boldsymbol{h}_t$ has a **short-term** memory.

- Together, they constitute all the long- and short-term memories that lead up to deliver the final output $\boldsymbol{h}_T$ at the last station.

If you are still thinking of Harry Potter, the truck is the Pensieve.

Using the cell state, LSTM becomes capable of preserving memories from the past. **But, why was this not possible with the RNNs before LSTM?**

Because the gradients vanish quickly for the $\boldsymbol{h}_t$'s. As a result, long-term memories do not persist. Differently, the cell states $\boldsymbol{c}_t$ in LSTM has stabilized gradient (discussed in § 5.2.8) and, thus, keeps the memory.

5.2.4 Cell Operations

An LSTM cell behaves like a living cell. It performs multiple operations to learn and preserve memories to draw inferences (the output).

Consider a cell operation in an LSTM layer in Figure 5.3. The cell processes one observation at a time in a timed sequence window $\{\boldsymbol{x}_{T-\tau}, \boldsymbol{x}_{T-\tau+1}, \ldots, \boldsymbol{x}_T\}$.

Suppose the cell is processing a time-step $\boldsymbol{x}_t$. The $\boldsymbol{x}_t$ flows into the cell as input, gets processed along the paths (in Figure 5.3) in the presence of the previous output $\boldsymbol{h}_{t-1}$ and cell state $\boldsymbol{c}_{t-1}$, and yields the updated output $\boldsymbol{h}_t$ and cell state $\boldsymbol{c}_t$.

The computations within the cell as implemented in TensorFlow from Jozefowicz, Zaremba, and Sutskever 2015 are given below in Equation 5.1a-5.1f,

$$i_t = \texttt{hard-sigmoid}(\boldsymbol{w}_i^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_i^{(h)}\boldsymbol{h}_{t-1} + b_i) \tag{5.1a}$$
$$o_t = \texttt{hard-sigmoid}(\boldsymbol{w}_o^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_o^{(h)}\boldsymbol{h}_{t-1} + b_o) \tag{5.1b}$$
$$f_t = \texttt{hard-sigmoid}(\boldsymbol{w}_f^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_f^{(h)}\boldsymbol{h}_{t-1} + b_f) \tag{5.1c}$$
$$\tilde{c}_t = \texttt{tanh}(\boldsymbol{w}_c^{(x)}\boldsymbol{x}_t + \boldsymbol{w}_c^{(h)}\boldsymbol{h}_{t-1} + b_c) \tag{5.1d}$$
$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \tag{5.1e}$$
$$h_t = o_t \texttt{tanh}(c_t) \tag{5.1f}$$

where,

- i_t, o_t, and f_t are input, output, and forget gates,
- `hard-sigmoid` is a segment-wise linear approximation of `sigmoid` function for faster computation. It returns a value between 0 and 1, defined as,

$$\texttt{hard-sigmoid}(x) = \begin{cases} 1 & , x > 2.5 \\ 0.2x + 0.5 & , -2.5 \leq x \leq 2.5 \\ 0 & , x < -2.5 \end{cases}$$

- $\tilde{c}_t$ is a temporary variable that holds the relevant information in the current time-step t,
- c_t and h_t are the cell state and outputs, and
- $\boldsymbol{w}_\cdot^{(x)}$, $\boldsymbol{w}_\cdot^{(h)}$, and $b_\cdot$ are the weight and bias parameters.

The intuition behind processing operations in Equation 5.1a-5.1f are broken into four steps and described below.

- **Step 1. Information.**

  The first step is to learn the information in the time-step input $\boldsymbol{x}_t$ alongside the cell's prior learned output $\boldsymbol{h}_{t-1}$. This learning is done

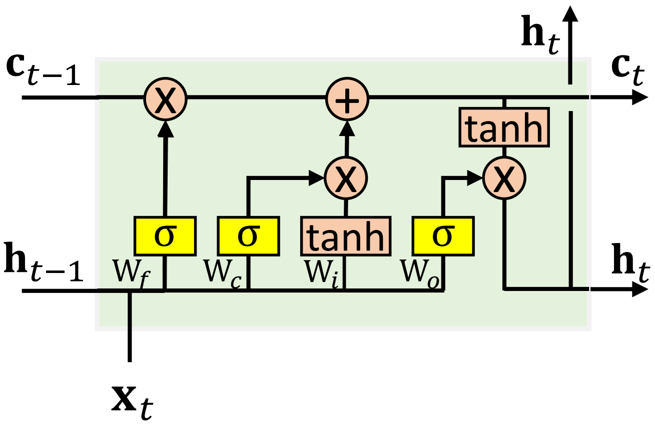
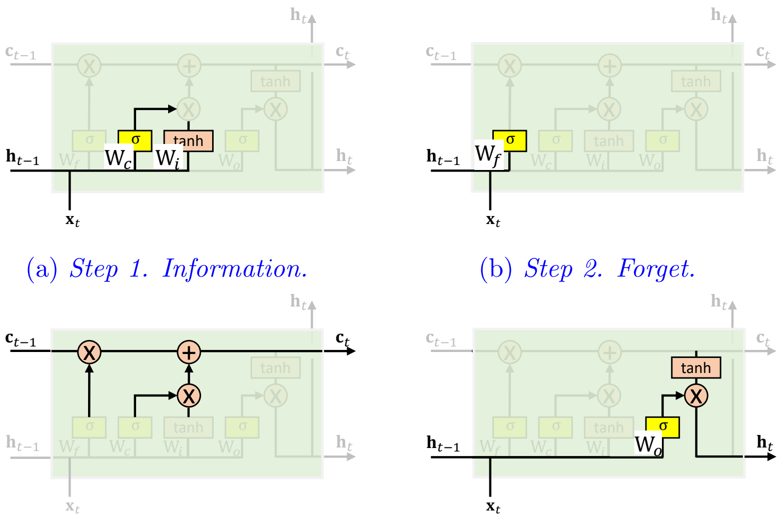


Figure 5.3. *The inside of an LSTM cell. The cell consists of three gates, input (i), output (o), and forget (f), made of sigmoid activation (σ) shown with yellow boxes. The cell derives relevant information through* `tanh` *activations shown with orange boxes. The cell takes the prior states ($\mathbf{c}_{t-1}$, $\mathbf{h}_{t-1}$), runs it through the gates, and draws information to yield the updated ($\mathbf{c}_t$, $\mathbf{h}_t$). Source Olah 2015.*



(a) *Step 1. Information.* (b) *Step 2. Forget.*

(c) *Step 3. Memory.* (d) *Step 4. Output.*

Figure 5.4. *Operations steps in an LSTM cell. Source Olah 2015.*

in two sub-steps given in Equation 5.1d and 5.1a and succinctly expressed below.

$$\tilde{c}_t = g\big(\boldsymbol{w}_c^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{w}_c^{(x)}\boldsymbol{x}_t + b_c\big)$$
$$i_t = \sigma\big(\boldsymbol{w}_i^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{w}_i^{(x)}\boldsymbol{x}_t + b_i\big)$$

The first equation finds the relevant information in $\boldsymbol{x}_t$. The equation applies a `tanh` activation. This activation has negative and positive values in $(-1, 1)$. The reason for using `tanh` is in step 3.

It is possible that $\boldsymbol{x}_t$ has information but it is redundant or irrelevant in the presence of the information already present with the cell from the previous $\boldsymbol{x}$'s.

To measure the relevancy, i_t is computed in the second equation. It is activated with `sigmoid` to have a value in $(0, 1)$. A value closer to 0 would mean the information is irrelevant and vice-versa.

- **Step 2. Forget.**

  Due to the new information coming in with $\boldsymbol{x}_t$, some of the previous memory may become immaterial. In that case, that memory can be *forgotten.*

  This forgetting decision is made at the forget gate in Equation 5.1c.

$$f_t = \sigma\big(\boldsymbol{w}_f^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{w}_f^{(x)}\boldsymbol{x}_t + b_f\big)$$

  The expression is `sigmoid` activated which yields an indicator between 0 and 1. If the indicator is close to zero, the past memory is forgotten. In this case, the information in $\boldsymbol{x}_t$ will replace the past memory.

  If the indicator is close to one, it means the memory is still pertinent and should be carried forward. But this does not necessarily indicate that the information in $\boldsymbol{x}_t$ is irrelevant or will not enter the memory.

- **Step 3. Memory.**

The previous steps find the information in $\boldsymbol{x}_t$, its relevance, and the need for the memory. These are concocted together to update the cell memory in Equation 5.1e.

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

The first component determines whether to carry on the memory from the past. If the forget gate is asking to forget, i.e., $f_t \to 0$, the cell's memory is bygone.

The second component in the equation, $i_t\tilde{c}_t$, is from step 1. This is a product of the relevance indicator, i_t, and the information, $\tilde{c}_t$. Ultimately, the product contains the relevant information found at t. Its value lies between -1 and $+1$. Depending on the different scenarios given in Table 5.1, they become close to zero or $|1|$.

As depicted in the table, the best scenario for $\boldsymbol{x}_t$ is when it has *relevant* information. In this case, both the magnitude of $|\tilde{c}_t|$ and i_t will be close to 1, and the resultant product will be far from 0. This scenario is at the bottom-right of the table.

The $i_t\tilde{c}_t$ component also makes the need for using a `tanh` activation in Equation 5.1d apparent.

Imagine using a positive-valued activation instead. In that case, $i_t\tilde{c}_t$ will be always positive. But since the $i_t\tilde{c}_t$ is added to state c_t in each iteration in $0, 1, \ldots, T$, a positive $i_t\tilde{c}_t$ will move c_t in an only positive direction. This can cause the state c_t to inflate.

On the other hand, `tanh` has negative to positive values in $(-1, 1)$. This allows the state to increase or decrease. And, keeps the state tractable.

Still, `tanh` is not mandatory. A positive-valued activation can also be used. This chapter implements `relu` activation in § 5.5 and 5.6, where it is found to work better than `tanh` when the input is scaled to Gaussian(0,1).

- **Step 4. Output.**

  At the last step, the cell output h_t is determined in Equation 5.1f. The output h_t is drawn from two components. One of them is the

Table 5.1. Scenarios of information present in a time-step $\boldsymbol{x}_t$ and its relevance in presence of the past memory.

		Information magnitude, $\lvert\tilde{c}_t\rvert$, close to,	
		0	**1**
Information relevance, i_t, close to,	**0**	No information in $\boldsymbol{x}_t$.	$\boldsymbol{x}_t$ has redundant information already present in the memory.
	1	Model (weights) inconsistent.	$\boldsymbol{x}_t$ has relevant new information.

output gate o_t that acts as a scale with value in $(0, 1)$. The other is a `tanh` activated value of the updated cell state c_t in step 3.

$$o_t = \sigma\left(\boldsymbol{w}_o^{(h)} \boldsymbol{h}_{t-1} + \boldsymbol{w}_o^{(h)} \boldsymbol{x}_t + b_o\right)$$
$$h_t = o_t \tanh(c_t)$$

The h_t expression behaves like short-term memory. And, therefore, h_t is also called a short state in TensorFlow.

These steps are shown in Figure 5.4a-5.4d. In each figure, the paths corresponding to a step are highlighted. Besides, the order of step 1 and 2 are interchangeable. But the subsequent steps 3 and 4 are necessarily in the same order.

5.2.5 Activations in LSTM

The activations in Equation 5.1d for $\tilde{c}_t$ and 5.1f for emitting h_t correspond to the `activation` argument in an `LSTM` layer in TensorFlow. By default, it is `tanh`. These expressions act as learned features and, therefore, can take any value. With `tanh` activation, they are in $(-1, 1)$. Other suitable activations can also be used for them.

On the other hand, the activations in Equation 5.1a-5.1c for input, output, and forget gates are referred to as the argument `recurrent_activation` in TensorFlow. These gates act as scales. Therefore, they are intended to stay in $(0, 1)$. Their default is, hence, `sigmoid`. For most purposes, it is essential to keep `recurrent_activation` as `sigmoid` (explained in § 5.2.8).

The ***`recurrent_activation`*** *should be* ***`sigmoid`****. The default* ***`activation`*** *is* ***`tanh`*** *but can be set to other activations such as* ***`relu`****.*

5.2.6 Parameters

Suppose an LSTM layer has m cells, i.e., the layer size equal to m. The cell mechanism illustrated in the previous section is for one cell in an LSTM layer. The parameters involved in the cell are, $\boldsymbol{w}_{\cdot}^{(h)}, \boldsymbol{w}_{\cdot}^{(x)}, b_{\cdot}$, where $\cdot$ is c, i, f, and o.

A cell intakes the prior output of all the other sibling cells in the layer. Given the layer size is m, the prior output from the layer cells will be an m-vector $\boldsymbol{h}_{t-1}$ and, therefore, the $\boldsymbol{w}_{\cdot}^{(h)}$ are also of the same length m.

The weight for the input time-step $\boldsymbol{x}_t$ is a p-vector given there are p features, i.e., $\boldsymbol{x}_t \in \mathbb{R}^p$. Lastly, the bias on a cell is a scalar.

Combining them for each of c, i, f, o, the total number of parameters in a cell is $4(m + p + 1)$.

In the LSTM layer, there are m cells. Therefore, the total number of parameters in a layer are,

$$n\_parameters = 4m(m + p + 1) \tag{5.2}$$

It is important to note here that **the number of parameters is independent of the number of time-steps processed by the cell**. That is, they are independent of the window size τ.

This implies that the parameter space does not increase if the window size is expanded to learn longer-term temporal patterns. While this might appear an advantage, in practice, the performance deteriorates after a certain limit on the window size (discussed more later in § 5.6.5).

An LSTM layer has $4m(m + p + 1)$ *parameters, where* m *is the size of the layer and* p *the number of features in the input.*

The number of LSTM parameters are independent of the sample window size.

5.2.7 Iteration Levels

A sample in LSTM is a window of time-step observations. Due to this, its iterations level shown in Figure 5.5 goes one level further than in MLPs (in Figure 4.14). In LSTMs, the iterations end at a time-step.

Within a time-step, all the cell operations described in § 5.2.4 are executed. But, unlike MLPs, the cell operations cannot be done directly with the tensor operation.

The cell operations in a time-step are sequential. The output of a time-step goes as an input to the next. Due to this, the time-steps are processed one-by-one. Furthermore, the steps within a time-step are also in order.

Because of the sequential operations, LSTM iterations are computationally intensive. However, samples within a batch do not interact in *stateless* LSTMs (default in TensorFlow) and, therefore, a batch is processed together using tensor operations.

5.2.8 Stabilized Gradient

The concept of keeping the memory from anywhere in the past was always present in RNNs. However, before LSTMs the RNN models were unable to learn long-term dependencies due to vanishing and exploding gradient issues.

```
epoch
 └─batch
    └─sample
       └─time-step
          ├─step-1 information
          ├─step-2 forget
          ├─step-3 memory
          └─step-4 output
```

Figure 5.5. *LSTM training iteration levels.*

This was achieved with the introduction of a cell state in LSTMs. The cell state stabilized the gradient. This section provides a simplified explanation behind this.

The output of a cell is h_T. Suppose the target is y_T, and we have a square loss function,

$$\mathcal{L}_T = (y_T - h_T)^2.$$

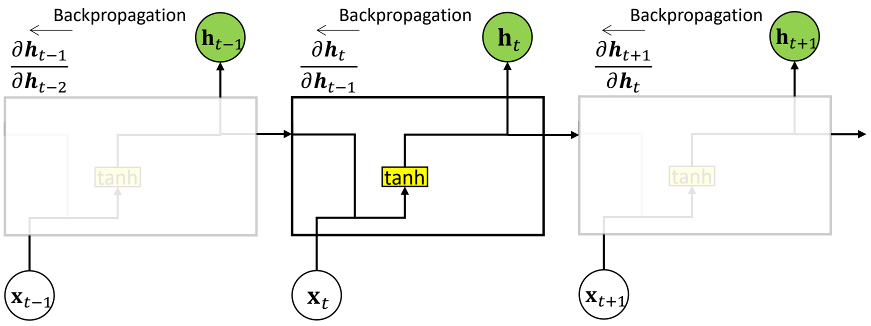
During the model estimation, the gradient of the loss with respect to a parameter is taken. Consider the gradient for a weight parameter,

$$\frac{\partial}{\partial w}\mathcal{L}_T = -2\underbrace{(y_T - h_T)}_{error}\frac{\partial h_T}{\partial w}.$$

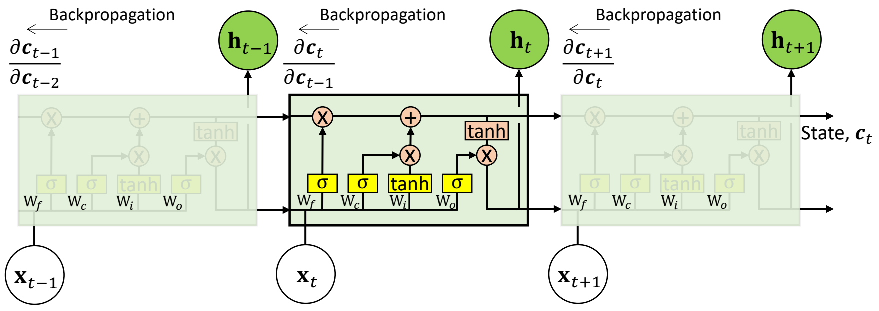
The term $(y_T - h_T)$ is the model error. During model training, the need is to propagate this error for model parameter update.

Whether the error appropriately propagates or not depends on the derivative $\frac{\partial h_T}{\partial w}$. If $\frac{\partial h_T}{\partial w}$ vanishes or explodes, the error gets distorted and model training suffers.

This was the case with the simple RNNs (refer to Figure 5.6a). In a simple RNN, there is no cell state. The cell output h_T is a function of the prior time-step output h_{T-1}.


Backpropagation
$\frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}}$
$\mathbf{h}_{t-1}$
tanh
$\mathbf{x}_{t-1}$
Backpropagation
$\frac{\partial \boldsymbol{h}_{t}}{\partial \boldsymbol{h}_{t-1}}$
$\mathbf{h}_{t}$
tanh
$\mathbf{x}_{t}$
Backpropagation
$\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_{t}}$
$\mathbf{h}_{t+1}$
tanh
$\mathbf{x}_{t+1}$

(a) *Simple RNN. The error propagates along the gradient of the hidden outputs* $\frac{h_t}{h_{t-1}}$ *which can explode or vanish.*


Backpropagation
$\frac{\partial \boldsymbol{c}_{t-1}}{\partial \boldsymbol{c}_{t-2}}$
$\mathbf{h}_{t-1}$
$\mathbf{x}_{t-1}$
Backpropagation
$\frac{\partial \boldsymbol{c}_{t}}{\partial \boldsymbol{c}_{t-1}}$
$\mathbf{h}_{t}$
$\mathbf{x}_{t}$
X
+
X
X
tanh
σ
σ
tanh
σ
W_f
W_c
W_i
W_o
Backpropagation
$\frac{\partial \boldsymbol{c}_{t+1}}{\partial \boldsymbol{c}_{t}}$
$\mathbf{h}_{t+1}$
$\mathbf{x}_{t+1}$
State, $\boldsymbol{c}_t$

(b) *LSTM backpropagation. The error propagates along the gradients of the cell states* $\frac{c_t}{c_{t-1}}$ *which are stable.*

Figure 5.6. *Illustration of backpropagation in a simple RNN (top) and LSTM (bottom).*

$$h_T \propto g(h_{T-1}).$$

Therefore, the derivative $\frac{\partial h_T}{\partial w}$ will be,

$$\frac{\partial h_T}{\partial w} \propto \frac{\partial h_T}{\partial h_{T-1}} \underbrace{\frac{\partial h_{T-1}}{\partial h_{T-2}} \cdots \frac{\partial h_{T-\tau+1}}{\partial h_{T-\tau}}}_{\text{can explode or vanish}} \frac{\partial h_{T-\tau}}{\partial w}. \tag{5.3}$$

As shown in Equation 5.3, the derivative $\frac{\partial h_T}{\partial w}$ is on the mercy of the chain product. A chain product is difficult to control. Since $\frac{\partial h_t}{\partial h_{t-1}}$ can take any value, the chain product can **explode** or **vanish**.

On the contrary, consider the LSTM Equations 5.1e-5.1f defined in § 5.2.4,

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$
$$h_t = o_t \texttt{tanh}(c_t).$$

Unlike a simple RNN, LSTM emits two outputs (refer to Figure 5.6b) in each time-step: a) a slow state c_t which is the cell state or the long-term memory, and b) a fast state h_t which is the cell output or the short-term memory.

Computing the derivative for LSTM from the expression $h_t = o_t \texttt{tanh}(c_t)$,

$$\frac{\partial h_T}{\partial w} \propto \frac{\partial h_T}{\partial c_T} \frac{\partial c_T}{\partial c_{T-1}} \frac{\partial c_{T-1}}{\partial c_{T-2}} \cdots \frac{\partial c_{T-\tau+1}}{\partial c_{T-\tau}} \frac{\partial c_{T-\tau}}{\partial w}. \tag{5.4}$$

Assume the forget gate f_t used in $c_t = f_t c_{t-1} + i_t \tilde{c}_t$ is inactive throughout the window $(T - \tau) : T$, i.e., $f_t = 1$, $t = T - \tau, \ldots, T$. Then,

$$c_t = c_{t-1} + i_t \tilde{c}_t.$$

In this scenario, $\frac{\partial c_t}{\partial c_{t-1}} = 1$. And, consequently, $\frac{\partial h_T}{\partial w}$ in Equation 5.4 becomes,

$$\frac{\partial h_T}{\partial w} \propto \frac{\partial h_T}{\partial c_T} \underbrace{\prod 1}_{\text{stabilizes the gradient}} \frac{\partial c_{T-\tau}}{\partial w} \tag{5.5}$$

This derivative in Equation 5.5 is now stable. It does not have a chain multiplication of elements that can take any value. Instead, it is replaced with a chain multiplication of 1's.

The cell state, thus, enables the error $(y_T - h_T)$ to propagate down the time-steps. In the scenario when the forget gate is closed $(f_t = 0)$ at some time-step t in $(T - \tau) : T$, the derivative becomes zero. This scenario means the memory/information before t is irrelevant. Therefore, learning/error propagation before t is not needed. In this case, the derivative desirably becomes zero.

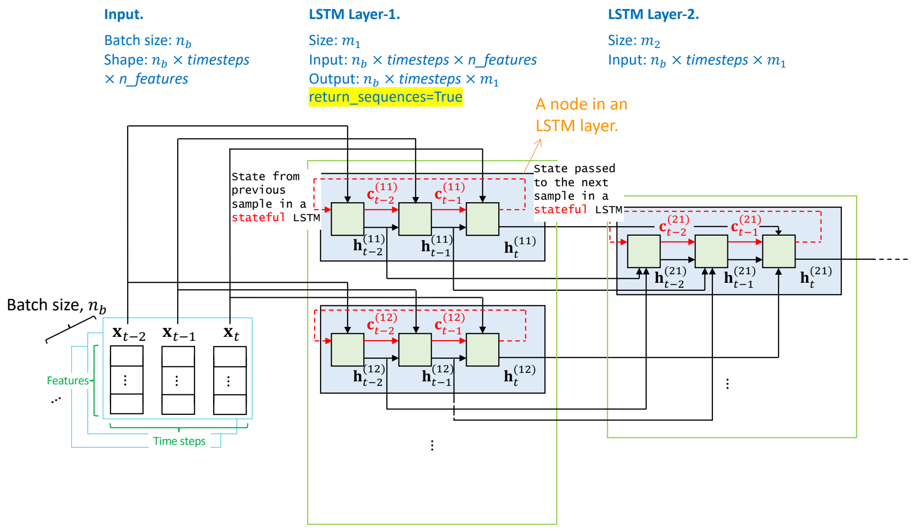
It is important to note that this stabilizing property in LSTM is achieved due to the additive auto-regression like $c_t = c_{t-1} + i_t \tilde{c}_t$, expression for the cell state. The additive expression makes the cell state the **long-term** memory.

The additive nature of the cell state c_t provides a stabilized gradient that enables the error to propagate down the time-steps. Due to this, long-term information can be preserved in c_t.

Besides, the stabilizing property is achieved if the forget gate takes value in $(0, 1)$. It is, therefore, essential that `recurrent_activation` is `sigmoid` as mentioned in § 5.2.5.

5.3 LSTM Layer and Network Structure

The previous section illustrated the LSTM fundamentals by deconstructing an LSTM cell in Figure 5.2. The cell input-output and the operations


Input.
Batch size: n_b
Shape: $n_b \times timesteps \times n\_features$
LSTM Layer-1.
Size: m_1
Input: $n_b \times timesteps \times n\_features$
Output: $n_b \times timesteps \times m_1$
return_sequences=True
LSTM Layer-2.
Size: m_2
Input: $n_b \times timesteps \times m_1$
A node in an LSTM layer.
State from previous sample in a stateful LSTM
State passed to the next sample in a stateful LSTM
Batch size, n_b
Features
Time steps
$\mathbf{x}_{t-2}$
$\mathbf{x}_{t-1}$
$\mathbf{x}_{t}$
$\mathbf{c}_{t-2}^{(11)}$
$\mathbf{c}_{t-1}^{(11)}$
$\mathbf{h}_{t-2}^{(11)}$
$\mathbf{h}_{t-1}^{(11)}$
$\mathbf{h}_{t}^{(11)}$
$\mathbf{c}_{t-2}^{(12)}$
$\mathbf{c}_{t-1}^{(12)}$
$\mathbf{h}_{t-2}^{(12)}$
$\mathbf{h}_{t-1}^{(12)}$
$\mathbf{h}_{t}^{(12)}$
$\mathbf{c}_{t-2}^{(21)}$
$\mathbf{c}_{t-1}^{(21)}$
$\mathbf{h}_{t-2}^{(21)}$
$\mathbf{h}_{t-1}^{(21)}$
$\mathbf{h}_{t}^{(21)}$

(a) *LSTM network input and hidden layers. The input is a batch of time-window of observations. This makes each sample in a batch a 2D array and the input batch a 3D array. The time-window is arbitrarily takes as three for illustration. The cells in blue boxes within the hidden LSTM layer is unwrapped to their time-step iterations shown with green boxes. The connected arcs show the transmission of time-indexed information between the layers. The first LSTM layer is emitting sequences (`LSTM(...,return_sequences=True)`). These sequences have the same notional time order as the input and are processed in the same order by the second LSTM layer. If the model is stateful, the cell state from the prior batch processing is preserved and accessed by the next batch.*

Figure 5.7. *LSTM Network Structure. Part I.*