Listing 7.18. The covariance of encoded features

```
encoder_model = Model(inputs=autoencoder.inputs,
                      outputs=autoencoder.get_layer(
                         'encoder').output)
encoded_features = np.array(encoder_model.predict(X)
   )
print('Encoded feature covariance\n',
      np.round(np.cov(encoded_features.T), 3))

# Encoded feature covariance
#   [[ 0.01 -0.     0.    -0.02]
#   [-0.     0.     0.    -0.02]
#   [ 0.     0.     0.02 -0.02]
#   [-0.02 -0.02 -0.02  4.36]]
```

7.8 Rules-of-thumb

- **Autoencoder Construction**.
  - Add unit-norm constraint on the weights. This prevents ill-conditioning of the model.
  - Add a linearly activated dense layer at the end of the encoder and decoder for calibration in most autoencoders.
  - The activation on the decoder output layer should be based on the range of the input. For example, `linear` if the input $\boldsymbol{x}$ is in $(-\infty, \infty)$ (scaled with `StandardScaler`) or `sigmoid` if $\boldsymbol{x}$ is in $(0, 1)$ (scaled with `MinMaxScaler`).
- **Sparse autoencoder**.
  - A sparsity constraint should be added to the encoder's output. Typically, it is a dense layer. The sparsity can be added as `activity_regularizer=tf.keras.regularizers.L1(l1=0.01)`.
  - The encoding size should be equal to the original data dimension (overcomplete). The sparsity penalty ensures that the encodings are useful and not trivial.
  - Sparse encodings are best suited for use in other tasks such as classification.

- **Denoising autoencoder**.
  - Unlike sparse autoencoders, denoising autoencoders regularize the decoder output to make them insensitive to minor changes in the input.
  - Train a denoising autoencoder by adding small gaussian noise to the input. Ensure that the loss function minimizes the difference of the original data $\boldsymbol{x}$ with the decodings of the noisy data $g(f(\boldsymbol{x}+\boldsymbol{\epsilon}))$ where $\boldsymbol{\epsilon}$ is Gaussian(0,σ).
  - They are useful for denoising or reconstruction objectives. But their encodings are typically not useful for classification tasks.
- **LSTM autoencoder**.
  - Use `tanh` activation in the LSTM layers in both encoder and decoder.
  - Works better for translation tasks, for example, English to Spanish text translation. Typically, they do not work well for data reconstruction.
- **Convolutional autoencoder**.
  - Encoder module has a stack of `Conv` and `Pooling` layers. They perform summarization of the useful features of the data.
  - Decoder module has a stack of `ConvTranspose` and `BatchNormalization` layers.
  - Decoder module should **not** have `Conv` or `Pooling` layers.

7.9 Exercises

1. At the beginning of the chapter it is mentioned that autoencoders were conceptualized with inspiration from an older concept of principal component analysis (PCA) in statistics.

   (a) A PCA model is linear. It was mentioned in § 4.2 that a dense layer network is equivalent to a linear model if the activations on each layer are linear (proved in Appendix A). Extending this to a dense autoencoder, does it become the same as a principal component analysis (PCA) model if the activations are linear? Refer to § 7.7.1.

   (b) Under what conditions an autoencoder becomes equivalent to PCA? Refer to § 7.2 and § 7.7.1.

2. In § 7.6.1 and 7.6.2 it is mentioned that a linearly activated dense layer should be added at the end of encoder and decoder modules.

   (a) What is the benefit of the linear dense layers?

   (b) When is it not essential?

   (c) Is it more essential in decoder than encoder? Explain.

3. The chapter provides examples of shallow autoencoders. In building a deep autoencoder what would you do in the following scenario?

   (a) In constructing a Sparse autoencoder would you consider adding sparsity penalty on the activations (output) of the intermediate layers in the encoder versus only on the last encoder layer?

   (b) An autoencoder is essentially a special case of a feed-forward network where both the predictors and the response are the same. Looking at an autoencoder from this perspective sometimes makes it difficult to separate encoder and decoder. Essentially, to tell the boundary. This becomes even harder in a deep network. How would you distinguish where an encoder ends and a decoder begins?

4. In § 7.7.1 a few aspects of regularizing autoencoder by constraining encoder and decoder are discussed. Based on them,

   (a) Should you consider regularizing the encoder or decoder functions f and g, respectively? Why?

   (b) If following the structure of the principal component of analysis, what is the difference in constraining weights on encoder and decoder to be unit-norm in a dense autoencoder? Why?

   (c) Why is it difficult to strictly enforce weights orthogonality in an autoencoder? Why is it present by default in PCA? Refer to § 7.7.3.

   (d) Orthogonal weights in PCA leads to independent features. However, an autoencoder even with orthogonal encoder weights does not guarantee independent encodings. Why? The answer to this question also answers Q 1a.

5. The goal of an autoencoder is to learn the essential properties of the data while training to reconstruct the input. There are different types of regularization available to improve learning. Broadly, regularization can be applied to either the encoding $f(\boldsymbol{x})$ or the decoding $g(f(\boldsymbol{x}))$. Refer to § 7.3 and answer the following.

   (a) Among them, when is regularizing the encoding $f(\boldsymbol{x})$ better?

   (b) When is regularizing the decoder $g(f(\boldsymbol{x}))$ better?

   (c) (Optional) Refer to § 14.2.1 in Goodfellow, Bengio, and Courville 2016 to show that a sparse autoencoder approximates a generative model and that the sparsity penalty arrives as a result of this framework.

6. (Optional) The encoder module in an autoencoder can be incorporated in a classifier network. There are several ways it can be incorporated. Appendix K provides a flexible implementation to try different approaches.

   (a) Run the model in the appendix to train a classifier by transferring the encoder weights learned in autoencoder training to a classifier.

(b) Make the transferred encoder weights trainable. Note the change in the number of trainable parameters compared to in Q 6a. Train the model.

Bibliography

[AB14] Guillaume Alain and Yoshua Bengio. “What regularized auto-encoders learn from the data-generating distribution”. In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 3563–3593.

[Adv06] Inc. Advanced Technology Services. *Downtime Costs Auto Industry $22k/Minute - Survey.* Mar. 2006. URL: `https://news.thomasnet.com/companystory/downtime-costs-auto-industry-22k-minute-survey-481017`.

[Bah57] RR Bahadur. “On unbiased estimates of uniformly minimum variance”. In: *Sankhyā: The Indian Journal of Statistics (1933-1960)* 18.3/4 (1957), pp. 211–224.

[Bas55] Dev Basu. “On statistics independent of a complete sufficient statistic”. In: *Sankhyā: The Indian Journal of Statistics (1933-1960)* 15.4 (1955), pp. 377–380.

[Bat+09] Iyad Batal et al. “Multivariate time series classification with temporal abstractions”. In: *Twenty-Second International FLAIRS Conference.* 2009.

[Bay+17] Inci M Baytas et al. “Patient subtyping via time-aware LSTM networks”. In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining.* 2017, pp. 65–74.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[Ben+13] Yoshua Bengio et al. "Generalized denoising auto-encoders as generative models". In: *Advances in neural information processing systems.* 2013, pp. 899–907.

[BGC02] Piero Bonissone, Kai Goebel, and Yu-To Chen. "Predicting wet-end web breakage in paper mills". In: *Working Notes of the 2002 AAAI symposium: Information Refinement and Revision for Decision Making: Modeling for Diagnostics, Prognostics, and Prediction.* 2002, pp. 84–92.

[BH89] Pierre Baldi and Kurt Hornik. "Neural networks and principal component analysis: Learning from examples without local minima". In: *Neural networks* 2.1 (1989), pp. 53–58.

[Bis+95] Christopher M Bishop et al. *Neural networks for pattern recognition.* Oxford university press, 1995.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning.* springer, 2006.

[Bou+11] Y-Lan Boureau, Nicolas Le Roux, et al. "Ask the locals: multi-way local pooling for image recognition". In: *2011 International Conference on Computer Vision.* IEEE. 2011, pp. 2651–2658.

[BPL10] Y-Lan Boureau, Jean Ponce, and Yann LeCun. "A theoretical analysis of feature pooling in visual recognition". In: *Proceedings of the 27th international conference on machine learning (ICML-10).* 2010, pp. 111–118.

[BS13] Pierre Baldi and Peter J Sadowski. "Understanding dropout". In: *Advances in neural information processing systems.* 2013, pp. 2814–2822.

[CB02] George Casella and Roger L Berger. *Statistical inference.* Vol. 2. Duxbury Pacific Grove, CA, 2002.

[Cho+14] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[CJK04] N Chawla, N Japkowicz, and A Kolcz. "Special issue on class imbalances". In: *SIGKDD Explorations* 6.1 (2004), pp. 1–6.

[CN11] Adam Coates and Andrew Y Ng. "Selecting receptive fields in deep networks". In: *Advances in neural information processing systems.* 2011, pp. 2528–2536.

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[Dai+17] Jifeng Dai et al. "Deformable convolutional networks". In: *Proceedings of the IEEE international conference on computer vision.* 2017, pp. 764–773.

[Den+09] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition.* Ieee. 2009, pp. 248–255.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of machine learning research* 12.Jul (2011), pp. 2121–2159.

[Elm90] Jeffrey L Elman. "Finding structure in time". In: *Cognitive science* 14.2 (1990), pp. 179–211.

[ESL14] Joan Bruna Estrach, Arthur Szlam, and Yann LeCun. "Signal recovery from pooling representations". In: *International conference on machine learning.* 2014, pp. 307–315.

[Fuk86] Kunihiko Fukushima. "A neural network model for selective attention in visual pattern recognition". In: *Biological Cybernetics* 55.1 (1986), pp. 5–15.

[Gam17] John Cristian Borges Gamboa. "Deep learning for timeseries analysis". In: *arXiv preprint arXiv:1701.01887* (2017).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[GFS07] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. "Multi-dimensional recurrent neural networks". In: *International conference on artificial neural networks.* Springer. 2007, pp. 549–558.

[GG16] Yarin Gal and Zoubin Ghahramani. “A theoretically grounded application of dropout in recurrent neural networks”. In: *Advances in neural information processing systems.* 2016, pp. 1019–1027.

[GŁ15] Tomasz Górecki and Maciej Łuczak. “Multivariate time series classification with parametric derivative dynamic time warping”. In: *Expert Systems with Applications* 42.5 (2015), pp. 2305–2312.

[GM02] Timothy J Gawne and Julie M Martin. “Responses of primate visual cortical V4 neurons to simultaneously presented stimuli”. In: *Journal of neurophysiology* 88.3 (2002), pp. 1128–1135.

[Gra12] Alex Graves. “Sequence transduction with recurrent neural networks”. In: *arXiv preprint arXiv:1211.3711* (2012).

[Gra13] Alex Graves. “Generating sequences with recurrent neural networks”. In: *arXiv preprint arXiv:1308.0850* (2013).

[GS05] Alex Graves and Jürgen Schmidhuber. “Framewise phoneme classification with bidirectional LSTM and other neural network architectures”. In: *Neural networks* 18.5-6 (2005), pp. 602–610.

[GS09] Alex Graves and Jürgen Schmidhuber. “Offline handwriting recognition with multidimensional recurrent neural networks”. In: *Advances in neural information processing systems.* 2009, pp. 545–552.

[GSC99] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. “Learning to forget: Continual prediction with LSTM”. In: (1999).

[GSS02] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. “Learning precise timing with LSTM recurrent networks”. In: *Journal of machine learning research* 3.Aug (2002), pp. 115–143.

[Gul+14] Caglar Gulcehre et al. “Learned-norm pooling for deep feedforward and recurrent neural networks”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases.* Springer. 2014, pp. 530–546.

[He+15a] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision.* 2015, pp. 1026–1034.

[He+15b] Kaiming He et al. "Spatial pyramid pooling in deep convolutional networks for visual recognition". In: *IEEE transactions on pattern analysis and machine intelligence* 37.9 (2015), pp. 1904–1916.

[He+16] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[Hin+12] Geoffrey Hinton et al. "COURSERA: Neural Networks for Machine Learning". In: *Lecture 9c: Using noise as a regularizer* (2012).

[Hir96] Hideo Hirose. "Maximum likelihood estimation in the 3-parameter Weibull distribution. A look through the generalized extreme-value distribution". In: *IEEE Transactions on Dielectrics and Electrical Insulation* 3.1 (1996), pp. 43–55.

[How+17] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction.* Vol. Second Edition. Springer Science & Business Media, 2009.

[HW62] David H Hubel and Torsten N Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of physiology* 160.1 (1962), p. 106.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[Jap+00] Nathalie Japkowicz et al. "Learning from imbalanced data sets: a comparison of various strategies". In: *AAAI workshop on learning from imbalanced data sets*. Vol. 68. Menlo Park, CA. 2000, pp. 10–15.

[JHD12] Yangqing Jia, Chang Huang, and Trevor Darrell. "Beyond spatial pyramids: Receptive field learning for pooled image features". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2012, pp. 3370–3377.

[Jor90] Michael I Jordan. "Attractor dynamics and parallelism in a connectionist sequential machine". In: *Artificial neural networks: concept learning*. 1990, pp. 112–127.

[JZS15] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. "An empirical exploration of recurrent network architectures". In: *International conference on machine learning*. 2015, pp. 2342–2350.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[Kla+17] Günter Klambauer et al. "Self-normalizing neural networks". In: *Advances in neural information processing systems*. 2017, pp. 971–980.

[Kob19a] Takumi Kobayashi. "Gaussian-Based Pooling for Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. 2019, pp. 11216–11226.

[Kob19b] Takumi Kobayashi. "Global feature guided local pooling". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 3365–3374.

[Kra16] Bartosz Krawczyk. "Learning from imbalanced data: open challenges and future directions". In: *Progress in Artificial Intelligence* 5.4 (2016), pp. 221–232.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[Lam+04] Ilan Lampl et al. “Intracellular measurements of spatial integration and the MAX operation in complex cells of the cat primary visual cortex”. In: *Journal of neurophysiology* 92.5 (2004), pp. 2704–2713.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), p. 436.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. “Network in network”. In: *arXiv preprint arXiv:1312.4400* (2013).

[LeC+90] Yann LeCun, Bernhard E Boser, et al. “Handwritten digit recognition with a back-propagation network”. In: *Advances in neural information processing systems.* 1990, pp. 396–404.

[LeC+98] Yann LeCun, Léon Bottou, et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[LGT16] Chen-Yu Lee, Patrick W Gallagher, and Zhuowen Tu. “Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree”. In: *Artificial intelligence and statistics.* 2016, pp. 464–472.

[LRR19] Francisco Louzada, Pedro L Ramos, and Eduardo Ramos. “A note on bias of closed-form estimators for the gamma distribution derived from likelihood equations”. In: *The American Statistician* 73.2 (2019), pp. 195–199.

[LS50] EL Lehmann and H Scheffé. “Completeness, Similar Regions, and Unbiased Estimation. I”. In: vol. 10. 4. JSTOR 25048038. 1950, 305fffdfffdfffd–340. DOI: `10.1007/978-1-4614-1412-4_23`.

[LS55] EL Lehmann and H Scheffé. “Completeness, Similar Regions, and Unbiased Estimation. II”. In: vol. 15. 3. JSTOR 25048243. 1955, pp. 219–236. DOI: `10.1007/978-1-4614-1412-4_24`.

[LWH90] Kevin J Lang, Alex H Waibel, and Geoffrey E Hinton. “A time-delay neural network architecture for isolated word recognition”. In: *Neural networks* 3.1 (1990), pp. 23–43.

[Ma+19] Chih-Yao Ma et al. "TS-LSTM and temporal-inception: Exploiting spatiotemporal dynamics for activity recognition". In: *Signal Processing: Image Communication* 71 (2019), pp. 76–87.

[Mal89] Stephane G Mallat. "A theory for multiresolution signal decomposition: the wavelet representation". In: *IEEE transactions on pattern analysis and machine intelligence* 11.7 (1989), pp. 674–693.

[Man15] James Manyika. *The Internet of Things: Mapping the value beyond the hype*. McKinsey Global Institute, 2015.

[MB05] Tom McReynolds and David Blythe. *Advanced graphics programming using OpenGL*. Elsevier, 2005.

[MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.

[Nag+11] Jawad Nagi et al. "Max-pooling convolutional neural networks for vision-based hand gesture recognition". In: *2011 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*. IEEE. 2011, pp. 342–347.

[NH10] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *ICML*. 2010.

[Ola15] Christopher Olah. "Understanding lstm networks". In: (2015).

[Ong17] Thuy Ong. *Facebook's translations are now powered completely by AI*. Aug. 2017. URL: https://www.theverge.com/2017/8/4/16093872/facebook-ai-translations-artificial-intelligence.

[OV10] Carlotta Orsenigo and Carlo Vercellis. "Combining discrete SVM and fixed cardinality warping distances for multivariate time series classification". In: *Pattern Recognition* 43.11 (2010), pp. 3787–3794.

[Pea01] Karl Pearson. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572.

[Pet+96] Thomas Petsche et al. “A neural network autoassociator for induction motor failure prediction”. In: *Advances in neural information processing systems.* 1996, pp. 924–930.

[PKT83] J Anthony Parker, Robert V Kenyon, and Donald E Troxel. “Comparison of interpolating methods for image resampling”. In: *IEEE Transactions on medical imaging* 2.1 (1983), pp. 31–39.

[Ran+07] Marc’Aurelio Ranzato, Christopher Poultney, et al. “Efficient learning of sparse representations with an energy-based model”. In: *Advances in neural information processing systems.* 2007, pp. 1137–1144.

[Ran+18] Chitta Ranjan et al. “Dataset: rare event classification in multivariate time series”. In: *arXiv preprint arXiv:1809.10717* (2018).

[Ran20] Chitta Ranjan. “Theory of Pooling”. In: *PrePrint, ResearchGate* (Nov. 2020). DOI: `10.13140/RG.2.2.23408.07688`. URL: `https://doi.org/10.13140/RG.2.2.23408.07688`.

[RBC08] Marc’Aurelio Ranzato, Y-Lan Boureau, and Yann L Cun. “Sparse feature learning for deep belief networks”. In: *Advances in neural information processing systems.* 2008, pp. 1185–1192.

[RF87] AJ Robinson and Frank Fallside. *The utility driven dynamic error propagation network.* University of Cambridge Department of Engineering Cambridge, 1987.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation.* Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[Rif+11a] Salah Rifai, Grégoire Mesnil, et al. “Higher order contractive auto-encoder”. In: *Joint European conference on machine learning and knowledge discovery in databases.* Springer. 2011, pp. 645–660.

[Rif+11b] Salah Rifai, Pascal Vincent, et al. "Contractive auto-encoders: Explicit invariance during feature extraction". In: *Icml*. 2011.

[RP98] Maximilian Riesenhuber and Tomaso Poggio. "Just one view: Invariances in inferotemporal cell tuning". In: *Advances in neural information processing systems*. 1998, pp. 215–221.

[RP99] Maximilian Riesenhuber and Tomaso Poggio. "Hierarchical models of object recognition in cortex". In: *Nature neuroscience* 2.11 (1999), pp. 1019–1025.

[Sae+18] Faraz Saeedan et al. "Detail-preserving pooling in deep networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 9108–9116.

[Sch12] Mark J Schervish. *Theory of statistics*. Springer Science & Business Media, 2012.

[Shi+16a] Wenzhe Shi, Jose Caballero, Ferenc Huszár, et al. "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 1874–1883.

[Shi+16b] Wenzhe Shi, Jose Caballero, Lucas Theis, et al. "Is the deconvolution layer the same as a convolutional layer?" In: *arXiv preprint arXiv:1609.07009* (2016).

[Smi16] Chris Smith. *iOS 10: Siri now works in third-party apps, comes with extra AI features*. June 2016. URL: `https://bgr.com/2016/06/13/ios-10-siri-third-party-apps/`.

[Smi85] Richard L Smith. "Maximum likelihood estimation in a class of nonregular cases". In: *Biometrika* 72.1 (1985), pp. 67–90.

[SP10] Thomas Serre and Tomaso Poggio. "A neuromorphic approach to computer vision". In: *Communications of the ACM* 53.10 (2010), pp. 54–61.

[SP97] Mike Schuster and Kuldip K Paliwal. "Bidirectional recurrent neural networks". In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[Spr+15] Jost Tobias Springenberg et al. "Striving for simplicity: The all convolutional net". In: *ICLR* (2015).

[Sri+14] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[SWK09] Yanmin Sun, Andrew KC Wong, and Mohamed S Kamel. “Classification of imbalanced data: A review”. In: *International Journal of Pattern Recognition and Artificial Intelligence* 23.04 (2009), pp. 687–719.

[SWP05] Thomas Serre, Lior Wolf, and Tomaso Poggio. “Object recognition with features inspired by visual cortex”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Vol. 2. Ieee. 2005, pp. 994–1000.

[SZ15] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *ICLR* (2015).

[Sze+15] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[Ten18] TensorFlow. *Standardizing on Keras: Guidance on High-level APIs in TensorFlow 2.0*. Dec. 2018. URL: `https://medium.com/tensorflow/standardizing-on-keras-guidance-on-high-level-apis-in-tensorflow-2-0-bad2b04c819a`.

[Ten19] TensorFlow. *What’s coming in TensorFlow 2.0*. Jan. 2019. URL: `https://medium.com/tensorflow/whats-coming-in-tensorflow-2-0-d3663832e9b8`.

[TM98] Carlo Tomasi and Roberto Manduchi. “Bilateral filtering for gray and color images”. In: *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*. IEEE. 1998, pp. 839–846.

[Vas+17] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[Vin+08] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, et al. "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 1096–1103.

[Vin+10] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, et al. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." In: *Journal of machine learning research* 11.12 (2010).

[VKE19] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. "Legendre Memory Units: Continuous-Time Representation in Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems*. 2019, pp. 15544–15553.

[Vog16] Werner Vogels. *Bringing the Magic of Amazon AI and Alexa to Apps on AWS*. Nov. 2016.

[Web+16] Nicolas Weber et al. "Rapid, detail-preserving image downscaling". In: *ACM Transactions on Graphics (TOG)* 35.6 (2016), pp. 1–6.

[WH18] Yuxin Wu and Kaiming He. "Group normalization". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 3–19.

[WL18] Travis Williams and Robert Li. "Wavelet pooling for convolutional neural networks". In: *International Conference on Learning Representations*. 2018.

[Wu+16] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[WZ95] Ronald J Williams and David Zipser. "Gradient-based learning algorithms for recurrent". In: *Backpropagation: Theory, architectures, and applications* 433 (1995).

[Xie+17] Saining Xie et al. "Aggregated residual transformations for deep neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1492–1500.

[Yan+09] Jianchao Yang et al. "Linear spatial pyramid matching using sparse coding for image classification". In: *2009 IEEE Conference on computer vision and pattern recognition*. IEEE. 2009, pp. 1794–1801.

[YC17] Zhi-Sheng Ye and Nan Chen. "Closed-form estimators for the gamma distribution derived from likelihood equations". In: *The American Statistician* 71.2 (2017), pp. 177–181.

[YK16] Fisher Yu and Vladlen Koltun. "Multi-scale context aggregation by dilated convolutions". In: *ICLR* (2016).

[Yu+14] Dingjun Yu et al. "Mixed pooling for convolutional neural networks". In: *International conference on rough sets and knowledge technology*. Springer. 2014, pp. 364–375.

[ZF13] Matthew D Zeiler and Rob Fergus. "Stochastic pooling for regularization of deep convolutional neural networks". In: *ICLR* (2013).

[ZS18] Zhilu Zhang and Mert Sabuncu. "Generalized cross entropy loss for training deep neural networks with noisy labels". In: *Advances in neural information processing systems*. 2018, pp. 8778–8788.

Appendix A

Importance of Nonlinear Activation

It is mentioned a few times in this book that a nonlinear activation is essential for the nonlinearity of a deep learning model. Specifically, this is emphasized in § 4.7 in Chapter 4. In this appendix, it is shown that a linear activation makes a multi-layer network a simple linear regression model.

In the following, Equation A.1 applies the activation successively from input to the output in an illustrative two-layer network in Chapter 4. It shows that if the activation is linear, i.e., $g(x) = x$, then any multi-layer network becomes equivalent to a linear model.

$$\begin{aligned}
\hat{y} &= g(\mathbf{w}^T \mathbf{z}^{(2)}) && \text{(A.1a)}\\
&= g(\mathbf{w}^T g(W^{(2)T} g(W^{(1)T}\mathbf{x}))) && \text{(A.1b)}\\
&= \mathbf{w}^T W^{(2)T} W^{(1)T}\mathbf{x}, \text{if activation } g \text{ is linear.} && \text{(A.1c)}\\
&= \tilde{W}^T \mathbf{x} && \text{(A.1d)}
\end{aligned}$$

where, $\tilde{W}^T = \mathbf{w}^T W^{(2)T} W^{(1)T}$.

Appendix B

Curve Shifting

Curve shifting is used to learn relationships between the variables at a certain time with an event at a different time. The event can be either from the past or the future.

For a rare event prediction, where the objective is to predict an event in advance, the event is shifted back in time. This approach is similar to developing a model to predict a transition state which ultimately leads to an event.

In Listing B.1, a user-defined function (UDF) for curve-shifting a binary response data is shown. In the UDF, an input argument `shift_by` corresponds to the time units we want to shift `y`. `shift_by` can be a positive or negative integer.

Listing B.1. Curve Shifting.

```
import numpy as np

def sign(x):
    return (1, -1)[x < 0]


def curve_shift(df, shift_by):
    '''
    This function will shift the binary labels in a
        dataframe.
    The curve shift will be with respect to the 1s.
    For example, if shift is -2, the following
        process
    will happen: if row n is labeled as 1, then
    - Make row (n+shift_by):(n+shift_by-1) = 1.
    - Remove row n.
    i.e. the labels will be shifted up to 2 rows up.

    Inputs:
    df       A pandas dataframe with a binary
        labeled column.
             This labeled column should be named as
                 'y'.
    shift_by An integer denoting the number of rows
        to shift.

    Output
    df       A dataframe with the binary labels
        shifted by shift.
    '''

    vector = df['y'].copy()
    for s in range(abs(shift_by)):
        tmp = vector.shift(sign(shift_by))
        tmp = tmp.fillna(0)
        vector += tmp
    labelcol = 'y'
    # Add vector to the df
    df.insert(loc=0, column=labelcol + 'tmp', value=
        vector)
    # Remove the rows with labelcol == 1.
    df = df.drop(df[df[labelcol] == 1].index)
    # Drop labelcol and rename the tmp col as
        labelcol
    df = df.drop(labelcol, axis=1)
    df = df.rename(columns={labelcol + 'tmp':
        labelcol})
    # Make the labelcol binary
    df.loc[df[labelcol] > 0, labelcol] = 1

    return df
```

`curve_shift` assumes the response is binary with `(0, 1)` labels, and for any row `t` where `y==1` it,

1. makes the `y` for rows `(t+shift_by):(t+shift_by-1)` equal to 1. Mathematically, this is $y_{(t-k):t} \leftarrow 1$, if $y_t = 1$ and k is the `shift_by`. And,

2. remove row `t`.

Step 1 shifts the curve. Step 2 removes the row when the event (sheet-break) occurred. As also mentioned in § 2.1.2, we are not interested in teaching the model to predict an event when it has already occurred.

The effect of the curve shifting is shown using Listing B.2.

Listing B.2. Testing Curve Shift.

```
import pandas as pd
import numpy as np

'''Download data here:
https://docs.google.com/forms/d/e/1
    FAIpQLSdyUk3lfDl7I5KYK_pw285LCApc-
    _RcoC0Tf9cnDnZ_TWzPAw/viewform
'''
df = pd.read_csv("data/processminer-sheet-break-rare
    -event-dataset.csv")
df.head(n=5)  # visualize the data.

# Hot encoding
hotencoding1 = pd.get_dummies(df['Grade&Bwt'])
hotencoding1 = hotencoding1.add_prefix('grade_')
hotencoding2 = pd.get_dummies(df['EventPress'])
hotencoding2 = hotencoding2.add_prefix('eventpress_'
    )

df = df.drop(['Grade&Bwt', 'EventPress'], axis=1)

df = pd.concat([df, hotencoding1, hotencoding2],
    axis=1)

df = df.rename(columns={'SheetBreak': 'y'})  #
    Rename response column name for ease of
    understanding
```

```
'''
Shift the data by 2 units, equal to 4 minutes.

Test: Testing whether the shift happened correctly.
'''
print('Before shifting')  # Positive labeled rows
    before shifting.
one_indexes = df.index[df['y'] == 1]
display(df.iloc[(one_indexes[0]-3):(one_indexes
    [0]+2), 0:5].head(n=5))

# Shift the response column y by 2 rows to do a 4-
    min ahead prediction.
df = curve_shift(df, shift_by = -2)

print('After shifting')  # Validating if the shift
    happened correctly.
display(df.iloc[(one_indexes[0]-4):(one_indexes
    [0]+1), 0:5].head(n=5))
```

The outputs of the listing are visualized in Figure 4.3 in Chapter 4.

Appendix C

Simple Plots

The result plots in every chapter are made using the definitions in Listing C.1.

Listing C.1. Simple plot definitions.

```
##############################
#####  Plotting functions #####
##############################

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

def plot_metric(model_history, metric,
    ylim=None, grid=False):
    sns.set()

    if grid is False:
        sns.set_style("white")
        sns.set_style("ticks")

    train_values = [
        value for key, value in model_history.items
            ()
        if metric in key.lower()
    ][0]
    valid_values = [
            value for key, value in model_history.items
                ()
            if metric in key.lower()
        ][1]

    fig, ax = plt.subplots()

    color = 'tab:blue'
    ax.set_xlabel('Epoch', fontsize=16)
    ax.set_ylabel(metric, color=color, fontsize=16)

    ax.plot(train_values, '--', color=color,
        label='Train ' + metric)
    ax.plot(valid_values, color=color,
        label='Valid ' + metric)
    ax.tick_params(axis='y', labelcolor=color)
    ax.tick_params(axis='both',
        which='major', labelsize=14)

    if ylim is None:
        ylim = [
            min(min(train_values),
                min(valid_values), 0.),
            max(max(train_values),
                max(valid_values))
        ]
    plt.yticks(np.round(np.linspace(ylim[0],
        ylim[1], 6), 1))
    plt.legend(loc='upper left', fontsize=16)

    if grid is False:
        sns.despine(offset=1, trim=True)

    return plt, fig


def plot_model_recall_fpr(model_history, grid=False)
    :
    sns.set()

    if grid is False:
        sns.set_style("white")
```

```python
        sns.set_style("ticks")

    train_recall = [
        value for key, value in model_history.items()
        if 'recall' in key.lower()
    ][0]
    valid_recall = [
        value for key, value in model_history.items()
        if 'recall' in key.lower()
    ][1]

    train_fpr = [
        value for key, value in model_history.items()
        if 'false_positive_rate' in key.lower()
    ][0]
    valid_fpr = [
        value for key, value in model_history.items()
        if 'false_positive_rate' in key.lower()
    ][1]

    fig, ax = plt.subplots()

    color = 'tab:red'
    ax.set_xlabel('Epoch', fontsize=16)
    ax.set_ylabel('value', fontsize=16)
    ax.plot(train_recall, '--', color=color, label='Train Recall')
    ax.plot(valid_recall, color=color, label='Valid Recall')
    ax.tick_params(axis='y', labelcolor='black')
    ax.tick_params(axis='both', which='major', labelsize=14)
    plt.legend(loc='upper left', fontsize=16)

    color = 'tab:blue'
    ax.plot(train_fpr, '--', color=color, label='Train FPR')
    ax.plot(valid_fpr, color=color, label='Valid FPR
        ')
    plt.yticks(np.round(np.linspace(0., 1., 6), 1))

    fig.tight_layout()
    plt.legend(loc='upper left', fontsize=16)

    if grid is False:
        sns.despine(offset=1, trim=True)

    return plt, fig
```

Appendix D

Backpropagation Gradients

Think of the two-layer neural network shown in Figure 4.2 illustrated in Chapter 4. We used a `binary_crossentropy` loss for this model shown in Equation 4.7. Without any loss of generality (w.l.o.g.), it can be expressed for a single sample as,

$$\mathcal{L}(\theta) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}) \tag{D.1}$$

where, $\hat{y}$ is the prediction for y, i.e. the $\Pr_\theta[y = 1]$ (denoted by p in Eq. 4.7) and θ is the set of all parameters $\{W^{(1)}, W^{(2)}, \mathbf{w}^{(o)}\}$. Here the bias parameters are assumed as 0 w.l.o.g.

The parameter update in an iterative estimation vary for different optimizers such as `adam` and `sgd` in TensorFlow. However, as they are all Gradient Descent based, the update rule generalizes as,

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta} \tag{D.2}$$

where η is a learning parameter.

As seen in the equation, the gradient guides the parameter estimation to reach its optimal value.

The gradient expression for a weight parameter can be derived as,

$$\frac{\partial \mathcal{L}}{\partial W^T} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^T} \tag{D.3}$$

A derivative of the weight transpose is used for mathematical convenience. Besides, $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ will always be the same and, therefore, can be ignored to express,

$$\frac{\partial \mathcal{L}}{\partial W^T} \propto \frac{\partial \hat{y}}{\partial W^T} \tag{D.4}$$

Additionally, the relationship between the layers' inputs and outputs are,

$$\hat{y} = \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)}) \tag{D.5a}$$
$$\mathbf{z}^{(2)} = g(W^{(2)T}\mathbf{z}^{(1)}) \tag{D.5b}$$
$$\mathbf{z}^{(1)} = g(W^{(1)T}\mathbf{x}) \tag{D.5c}$$

where σ is the activation on the output layer and g on the hidden layers. Note that g can be different across layers but shown to be the same here for simplicity.

Using Equation D.4-D.5, we can express the gradients for each weight parameter as,

Output Layer.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(o)T}} \propto \frac{\partial \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)})}{\partial \mathbf{w}^{(o)T}} \tag{D.6}$$

Hidden Layer-2.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W^{(2)T}} &\propto \frac{\partial \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)})}{\partial W^{(2)T}} \\ &\propto \frac{\partial}{\partial W^{(2)T}}\sigma(\mathbf{w}^{(o)T}g(W^{(2)T}\mathbf{z}^{(1)})) \\ &\propto \frac{\partial \sigma(\mathbf{w}^{(o)T}g(W^{(2)T}\mathbf{z}^{(1)}))}{\partial g(W^{(2)T}\mathbf{z}^{(1)})}\frac{\partial g(W^{(2)T}\mathbf{z}^{(1)})}{\partial W^{(2)T}} \end{aligned} \tag{D.7}$$

Hidden Layer-1.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W^{(1)T}} &\propto \frac{\partial \sigma(\mathbf{w}^{(o)T}\mathbf{z}^{(2)})}{\partial W^{(1)T}} \\ &\propto \frac{\partial}{\partial W^{(1)T}}\sigma(\mathbf{w}^{(o)T}g(W^{(2)T}g(W^{(1)T}\mathbf{x}))) \\ &\propto \frac{\partial \sigma(\mathbf{w}^{(o)T}g(W^{(2)T}g(W^{(1)T}\mathbf{x})))}{\partial g(W^{(2)T}g(W^{(1)T}\mathbf{x}))}\frac{\partial g(W^{(2)T}g(W^{(1)T}\mathbf{x}))}{\partial g(W^{(1)T}\mathbf{x})}\frac{\partial g(W^{(1)T}\mathbf{x})}{\partial W^{(1)T}} \end{aligned} \tag{D.8}$$

Appendix E

Data Temporalization

Temporal models such as LSTM and convolutional networks are a bit more demanding than other models. A significant amount of time and attention goes into preparing the data that fits them.

First, we will create the three-dimensional tensors of shape: (*samples*, *timesteps*, *features*) in Listing E.1. *Samples* mean the number of data points. *Timesteps* is the number of time steps we look back at any time t to make a prediction. This is also referred to as the `lookback` period. The *features* are the number of features the data has, in other words, the number of predictors in multivariate data.

Listing E.1. Data temporalization

```
def temporalize(X, y, lookback):
    '''
    Inputs
    X           A 2D numpy array ordered by time of
       shape: (n_observations x n_features)
    y           A 1D numpy array with indexes aligned
       with X, i.e. y[i] should correspond to X[i].
        Shape: n_observations.
    lookback   The window size to look back in the
       past records. Shape: a scalar.

    Output
    output_X   A 3D numpy array of shape: ((
       n_observations-lookback-1) x lookback x
```

```
        n_features)
    output_y   A 1D array of shape: (n_observations-lookback-1), aligned with X.
    '''
    output_X = []
    output_y = []
    for i in range(len(X) - lookback - 1):
        t = []
        for j in range(1, lookback + 1):
            # Gather the past records upto the lookback period
            t.append(X[[(i + j + 1)], :])
        output_X.append(t)
        output_y.append(y[i + lookback + 1])
    return np.squeeze(np.array(output_X)), np.array(output_y)


def flatten(X):
    '''
    Flatten a 3D array.

    Input
    X            A 3D array for lstm, where the array is sample x timesteps x features.

    Output
    flattened_X   A 2D array, sample x features.
    '''
    flattened_X = np.empty(
        (X.shape[0], X.shape[2]))  # sample x features array.
    for i in range(X.shape[0]):
        flattened_X[i] = X[i, (X.shape[1] - 1), :]
    return flattened_X


def scale(X, scaler):
    '''
    Scale 3D array.

    Inputs
    X               A 3D array for lstm, where the
        array is sample x timesteps x features.
    scaler          A scaler object, e.g., sklearn.
        preprocessing.StandardScaler, sklearn.
        preprocessing.normalize

    Output
    X               Scaled 3D array.
    '''
    for i in range(X.shape[0]):
        X[i, :, :] = scaler.transform(X[i, :, :])

    return X
```

Additional helper functions, `flatten()` and `scale()`, are defined to make it easier to work with the tensors.

Testing

Since temporalization is an error-prone transformation, it is important to test the input tensors as shown in Listing E.2.

Listing E.2. Testing data temporalization.

```
"""### Temporalized data scale testing"""

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Sort by time and drop the time column.
df['DateTime'] = pd.to_datetime(df.DateTime)
df = df.sort_values(by='DateTime')
df = df.drop(['DateTime'], axis=1)

input_X = df.loc[:, df.columns != 'y'].values  #
    converts df to numpy array
input_y = df['y'].values

n_features = input_X.shape[1]  # number of features

# Temporalize the data
lookback = 5
```

```python
X, y = temporalize(X=input_X,
                   y=input_y,
                   lookback=lookback)

X_train, X_test, y_train, y_test = train_test_split(
    np.array(X),
    np.array(y),
    test_size=0.2,
    random_state=123)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train,
    y_train,
    test_size=0.2,
    random_state=123)

# Initialize a scaler using the training data.
scaler = StandardScaler().fit(flatten(X_train))

X_train_scaled = scale(X_train, scaler)

'''
Test: Check if the scaling is correct.

The test succeeds if all the column means
and variances are 0 and 1, respectively, after
flattening.
'''
print('==== Column-wise mean ====\n', np.mean(flatten(X_train_scaled), axis=0).round(6))
print('==== Column-wise variance ====\n', np.var(flatten(X_train_scaled), axis=0))

# ==== Column-wise mean ====
#  [-0.  0.  0. -0. -0. -0. -0.  0. -0. -0.  0. -0. -0.  0.  0.  0.  0.  0.
#  -0. -0. -0. -0.  0.  0. -0. -0.  0.  0. -0.  0.  0.  0.  0.  0. -0.  0.
#   0.  0. -0.  0.  0. -0. -0.  0. -0.  0.  0.  0.  0. -0. -0. -0.  0.  0.
#   0.  0.  0. -0. -0.  0. -0. -0. -0. -0.  0.  0. -0.  0.  0.]
# ==== Column-wise variance ====
#  [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
    1. 1. 1. 1. 1. 1. 1. 1.
#   1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
    1. 1. 1. 1. 1. 1. 1. 1.
#   1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
    1. 1. 1. 0. 0.]
```

The temporalization output is shown in Figure 5.8 in Chapter 5.

Besides, the `scale()` function is tested and the outputs are shown in the listing. As expected, the mean and variances become 0 and 1 after using a `StandardScaler()` on the temporalized data.

Appendix F

Stateful LSTM

A typical *stateless* LSTM cell illustrated in Chapter 5 processes only as much "past" in the data as defined by the timesteps. This could be restrictive because:

> "Ideally, we want to expose the network to the entire sequence and let it learn the inter-dependencies, rather than we define those dependencies explicitly in the framing of the problem.
>
> ...
>
> This is truly how the LSTM networks are intended to be used."
>
> – Jason Brownlee

It is, in fact, learned in temporal modeling with LSTMs and convolutional neural networks in Chapter 5 and 6 that a larger lookback (timesteps) typically improves a model's accuracy. This is logical because there are lagged dependencies. And, a larger timestep allows the model to look farther in the past to have more context for prediction.

If going farther back in time improves accuracy then can we go all the way in the past? The answer is yes. This can be done with *stateful* LSTMs.

Using a stateful LSTM is a simple approach. This approach requires the input samples to be ordered by time. Also, unlike typical model fitting that resets the model every training iteration, with the stateful LSTM it is reset every epoch.

The implementation in this appendix shows that a stateful LSTM cell processes the entire input training data sequentially and learns the dependencies from anywhere in the past.

However, as lucrative as the stateful LSTM appears, it does not always work. This approach tends to work better if the data is stationary. For example, text documents. The writing pattern does not change significantly and therefore, the process is stationary.

But most time series processes are non-stationary. The dependencies in them are confounded due to the non-stationarity. Therefore, a window of timesteps in which the process is assumed to be stationary tends to work better. For the same reason, a large time step should be carefully chosen in non-stationary processes.

Stateful LSTM is suitable if the data is stationary, i.e., the patterns do not change with time.

Implementing a stateful LSTM is different from traditional models. In the following, the implementation steps are given.

Data Preparation

In a stateful LSTM network, it is necessary to have the size of the input data as a multiple of the batch size. The data preparation is thus slightly different. In Listing F.1 the number of samples for train, valid, and test is taken as a multiple of the batch size which is closest to their original size.

Listing F.1. Stateful LSTM model data preparation.

```python
# Time ordered original data.
lookback_stateful = 1
# Temporalize the data
X, y = temporalize(X=input_X,
                   y=input_y,
                   lookback=lookback_stateful)

batch_size = 128

# Train, valid and test size set
# to match the previous models.
train_size = 13002
valid_size = 3251
test_size = 3251

X_train_stateful, y_train_stateful =
    np.array(
        X[0:int(train_size / batch_size) *
            batch_size]),
            np.array(
        y[0:int(train_size / batch_size) *
            batch_size])
X_valid_stateful, y_valid_stateful = np.array(
    X[int(train_size / batch_size) *
      batch_size:int((train_size + valid_size) /
          batch_size) *
      batch_size]), np.array(
          y[int(train_size / batch_size) *
            batch_size:int((train_size +
                valid_size) / batch_size) *
            batch_size])
X_test_stateful, y_test_stateful = np.array(
    X[int((train_size + test_size) / batch_size) *
        batch_size:]), np.array(
        y[int((train_size + test_size) /
            batch_size) * batch_size:])

X_train_stateful =
    X_train_stateful.reshape(
        X_train_stateful.shape[0],
        lookback_stateful,
        n_features)
X_valid_stateful =
    X_valid_stateful.reshape(
        X_valid_stateful.shape[0],
        lookback_stateful,
        n_features)
X_test_stateful =
    X_test_stateful.reshape(
        X_test_stateful.shape[0],
        lookback_stateful,
        n_features)

scaler_stateful =
    StandardScaler().fit(flatten(
        X_train_stateful))

X_train_stateful_scaled =
    scale(X_train_stateful,
          scaler_stateful)

X_valid_stateful_scaled =
    scale(X_valid_stateful,
          scaler_stateful)
X_test_stateful_scaled =
    scale(X_test_stateful,
          scaler_stateful)
```

The question is, why the batch size is required in a stateful model?

It is because when the model is stateless, TensorFlow allocates a tensor for the states of size `output_dim` based on the number of LSTM cells. At each sequence processing, this state tensor is reset.

On the other hand, TensorFlow propagates the previous states for each sample across the batches in a stateful model. In this case, the structure to store the states is of shape `(batch_size, output_dim)`. Due to this, it is necessary to provide the batch size while constructing the network.

Stateful Model

A stateful LSTM model is designed to traverse the entire past in the data for the model to self-learn the distant inter-dependencies instead of limiting it in a lookback window. This is achieved with a specific training procedure shown in Listing F.2.

The LSTM layer is made stateful by setting its argument `stateful=True`.

Listing F.2. Stateful LSTM model.

```python
# Stateful model.

timesteps_stateful =
    X_train_stateful_scaled.shape[1]
n_features_stateful =
    X_train_stateful_scaled.shape[2]

model = Sequential()
model.add(
    Input(shape=(timesteps_stateful,
                 n_features_stateful),
          batch_size=batch_size,
          name='input'))
model.add(
    LSTM(8,
         activation='relu',
         return_sequences=True,
         stateful=True,
         name='lstm_layer_1'))
model.add(Flatten())
model.add(Dense(units=1,
                activation='sigmoid',
                name='output'))

model.summary()

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

Unlike stateless LSTM, the cell states are preserved at every training iteration in a stateful LSTM. This allows it to learn the dependencies between the batches and, therefore, long-term patterns in significantly long sequences. However, we do not want the state to be transferred from one epoch to the next. To avoid this, we have to manually reset the state after each epoch.

A custom operation during training iterations can be performed by overriding the definitions in `tf.keras.callbacks.Callback`[1]. The `Callback()` class has definitions to perform operations at the beginning and/or end of a `batch` or `epoch` for both `test` and `train`. Since we require to reset the model states at the end of every epoch, we override the `on_epoch_end()` in Listing F.3 with `model.reset_states()`.

Listing F.3. Custom Callback() for Stateful LSTM model.

```
class ResetStatesCallback(
    tf.keras.callbacks.Callback):

    def on_epoch_end(self, epoch, logs={}):
        self.model.reset_states()
```

We now train the model in Listing F.4. In the `model.fit()`, we set the argument `callbacks` equal to our custom defined `ResetStatesCallback()`. Also, we set `shuffle=False` to maintain the time ordering of the samples during the training.

Listing F.4. Stateful LSTM model fitting.

```
history = model.fit(
    x=X_train_stateful_scaled,
    y=y_train_stateful,
    callbacks=[ResetStatesCallback()],
    batch_size=batch_size,
    epochs=100,
    shuffle=False,
    validation_data=(X_valid_stateful_scaled,
                     y_valid_stateful),
    verbose=0).history
```

The results from stateful LSTM on the sheet-break time series is poorer than the other LSTM models in Chapter 5. As alluded to earlier, a potential reason is that the process is non-stationary. Due to this, the dependencies change over time and are difficult to learn.

[1] /api_docs/python/tf/keras/callbacks/Callback

Appendix G

Null-Rectified Linear Unit

Rectified Linear Units (`relu`) is one of the most common activation functions. It is expressed as,

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

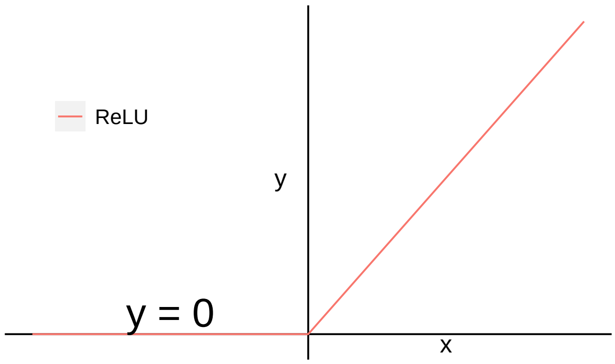
As shown in the equation, `relu` makes any non-positive x as 0. This brings nonlinearity to the model but at the cost of feature manipulation.

Such manipulation affects the pooling operation in convolutional networks (see § 6.12.2 in Chapter 6). The manipulation distorts the original distribution of the features which makes some pooling statistics, e.g., average, inefficient.

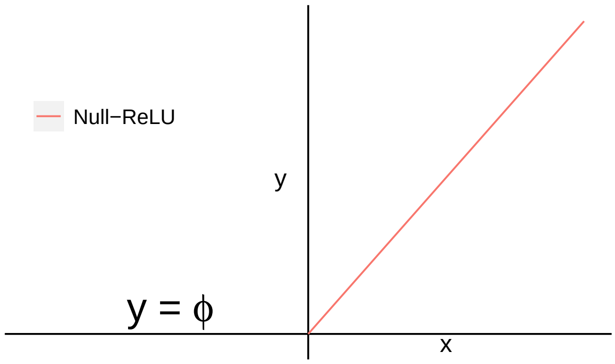
A resolution mentioned in § 6.13.3 is replacing `relu` activation with null-`relu`, which is defined as,

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ \phi, & \text{otherwise} \end{cases} \tag{G.2}$$

where ϕ denotes *null*. Unlike Equation G.1, null-`relu` acts as dropping the non-positive x's instead of treating them as 0. Both are visualized in Figure G.1a and G.1b, respectively.



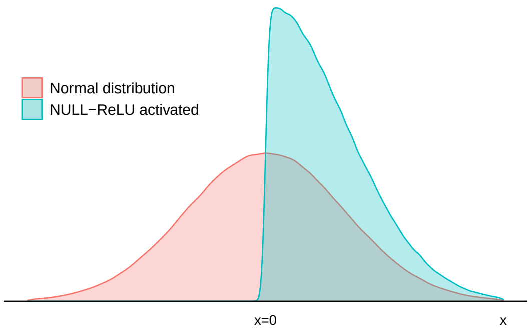
(a) *Traditional **`relu`** activation.*



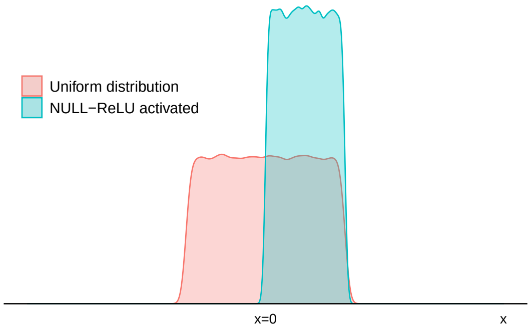
(b) *A variant null-**`relu`** activation.*

Figure G.1. *The traditional Rectified Linear Unit (**`relu`**) activation (top) transforms any non-positive x to zero. This is a nonlinear operation essential in most deep learning layers. However, in between a convolutional and pooling operation, the **`relu`** transformation can have an unwanted effect. The pooling attempts to draw a summary statistic from convolution output. But **`relu`** brings artificial* 0*'s that subvert summary information. A Null-**`relu`** activation (bottom) mitigates this by replacing the non-positive x's with ϕ (null). Unlike* 0*s in **`relu`**, nulls do not add any artificial information; they only mask the non-positive x's.*

Null-`relu`'s impact on the activated features for normal and uniform distributions are shown in Figure G.2a and G.2b, respectively. As opposed to the `relu` effect shown in Figure 6.28a and 6.28b, the activated feature distributions are still known—half-gaussian and uniform. Therefore, efficient pooling statistics such as in Kobayashi 2019a can be used.



(a) *Normal Distribution after null-`relu` activation.*



(b) *Uniform Distribution after null-`relu` activation.*

Figure G.2. *The distribution of the feature map is distorted by a `relu` activation. While any nonlinear activation distorts the original distribution, `relu`'s is severe because it artificially adds zeros for every non-positive x. Due to this, the activated x's distribution becomes extremely heavy at zero. Such distributions do not belong to known or well-defined distribution families. A variant of `relu` called null-`relu` transforms the non-positive x to null (ϕ). This is equivalent to throwing the non-positive x's instead of assuming them to be zero. If the original distribution is normal (top) or uniform (bottom), the null-`relu` activated are half-Gaussian and Uniform, respectively. Therefore, it has a less severe effect on distribution.*