JR Kargbo
April 28, 2023
Deep Learning

Twitter Sentiment Analysis using Convolutional Neural Networks

Twitter is one of the world's largest social networks, with it having a user base of 238 million monetizable daily active users as of Q2 2022 [1]. This makes it a potentially useful tool for organizations to attempt to gain information about how people feel about various topics, people, and groups. However, there are various challenges in doing this task. For one thing, manually determining the sentiment of every post you come across is going to be quite time consuming. Additionally, detecting emotions and sentiment from just text is difficult even for people, let alone computers. Finally, Twitter posts have famously had strict character limits, with an initial limit of 140 characters, increased to 280 in 2017, and Twitter Blue users recently being allowed to post messages of up to 4000 characters. This means that there might not be as much information to work off of compared to other forms of communication.

However, recent advances in Machine Learning have provided potential advances in this field. Convolutional Neural Networks and other Natural Language Processing tools could be incredibly good at automating this classification process. Using a set of Twitter Data collected from Kaggle, I've developed a model using a Convolutional Neural Network, and trained a BERT (Bidirectional Encoder Representations from Transformer) model that should both provide insights into classifying this data.

The dataset I'm using for this project is a Twitter Sentiment Analysis dataset from Kaggle, a repository where people can upload, work on, and discuss datasets, models, and code on a wide range of topics and tasks. This dataset has two files: A 'training set' with 74,682 rows of data in it that I've randomly split 80/20 into my training set and validation set, and a 'validation set' with 1000 rows of data that I'm instead using as my test set. The data also has Entity (Topic) information for if we want to do an Entity-Based classification. However, my models aren't taking the topic into account, and are only using the messages themselves to model. There are 4 different responses that a message in the data can have: Positive, Negative, Neutral, or Irrelevant. However, Irrelevant is only really applicable in an Entity-level analysis, and in our network, we're only working off of the information given in the message itself. As such, only Positive, Negative, and Neutral outcomes are relevant to this analysis, so I've removed all records in the data marked as Irrelevant. This leaves me with 49347 rows of training data, 12345 rows of validation data, and 828 rows of test data.

Before we can actually use our data in a model, however, we first have to properly process the data into a form that can be read by our networks. By default, these messages will have a lot of irrelevant information in them that can be removed. Additionally, most convolutional neural networks can only take numeric data as input, and any strings will have to be converted into some numeric form. The first step I took in processing the data was removing any punctuation and extra symbols from the data, as these were likely to have a limited impact on the sentiment of a message. I did this using a regular expression to filter out everything except capital and lowercase characters. I also set all text in the file to lower case.

Next, we remove any Stop Words from the data. Stop Words refers to any full words that don't add much information from the data. This includes words such as "and", "is", or "the". The Natural Language Toolkit (or NLTK), a python library that has a large amount of tools to help with Natural Language Processing, has tools to help remove Stop Words from a piece of text.

After this, the next step is to Lemmatize or Stem the words in the text. This refers to converting a word into its base form. For example, turning the word 'exercising' into 'exercise'. The difference between the two is in their methodology. Stemmers generally use a heuristic approach to converting a word to its base form, resulting in some odd results in some cases. Lemmatizers, meanwhile, use a dictionary to help try to perform this task better [2]. NLTK, again, has built-in tools for both Stemming and Lemmatizing text. I decided to use a Lemmatizer over a Stemmer.

Next, we need to Tokenize our data. This is the point in the process where we finally convert our data into a numeric form that can be processed by a neural network. There are many different tools we can use to perform this task. However, the method that I've decided to go with is the One-Hot method in Keras's pre-processing library. This function uses hashing to convert our dataset into numbers, with the user having to set how many spaces are available in the hash table. Since this uses a hash function, there's the risk of collisions, or having multiple words be stored in the same space in the hash table, causing them to be considered the same word by our model [3]. In order to reduce the likelihood of this occurring, I've chosen 10000 as my vocabulary size.

Finally, Neural Networks generally expect input of a consistent size, so I added padding to each message to ensure that they're all a consistent length. In this case, I set them to 163, as this was the largest message I saw I had in the data.

Now after all of this extensive pre-processing, it's finally time to create our classification model. The model I've created is a Convolutional Neural Network. Neural Networks are models that take in input, send that input through a series of nodes, each of which have their own weights and biases, and then return a final output. What makes them special, however, is their ability to adjust these weights and biases as it works over the data more. More specifically, it uses a method called Backpropagation, which involves using calculus to find the weights and biases that minimize a predefined loss function, and then using the results of this to update the weights and biases of our network [4]. Convolutional Neural Networks take this a step further, by filtering the data to try and discover patterns that may exist [5].

However, before we go into the Convolutional Layers, we need to go over the Embedding Layer. This is a Layer that converts our one-hot encoded inputs into a set of dense vectors that can be trained by our model. This reduces the dimensionality of our data, and also allows us to find relationships between each item in our dataset [13].

Next, let's go over some of the other layers in our CNN, such as the Convolutional Layers, which filter out data, Pooling Layers, which attempts to reduce the dimensionality of our data by summarizing it, and Activation Layers, which add in a non-linear element to our model [5]. In my model, I have 4 Convolutional Layers, each of which have 128, 256, 256, and 128 nodes in them respectively. Each of these convolutional layers have a stride of kernel size of 3, referring to how large the filter we're using is. Each of these layers also have pooling layers, and a Leaky ReLU activation function. Leaky ReLu was chosen as the activation layer because it helps to solve both the 'vanishing/exploding' gradient problems of other activation functions, and the 'dying ReLU' problems of standard 'ReLU'. Vanishing/Exploding Gradients are a problem where gradients can either grow or shrink exponentially. Meanwhile, Dying ReLu is a problem where neurons could potentially die out during training [6]. Additionally, each of these layers have batch normalization layers to normalize their output and reduce the effects of outliers in the data, and dropout layers to randomly turn off the output of certain nodes [7], making the rest of the model more robust and preventing overfitting [8].

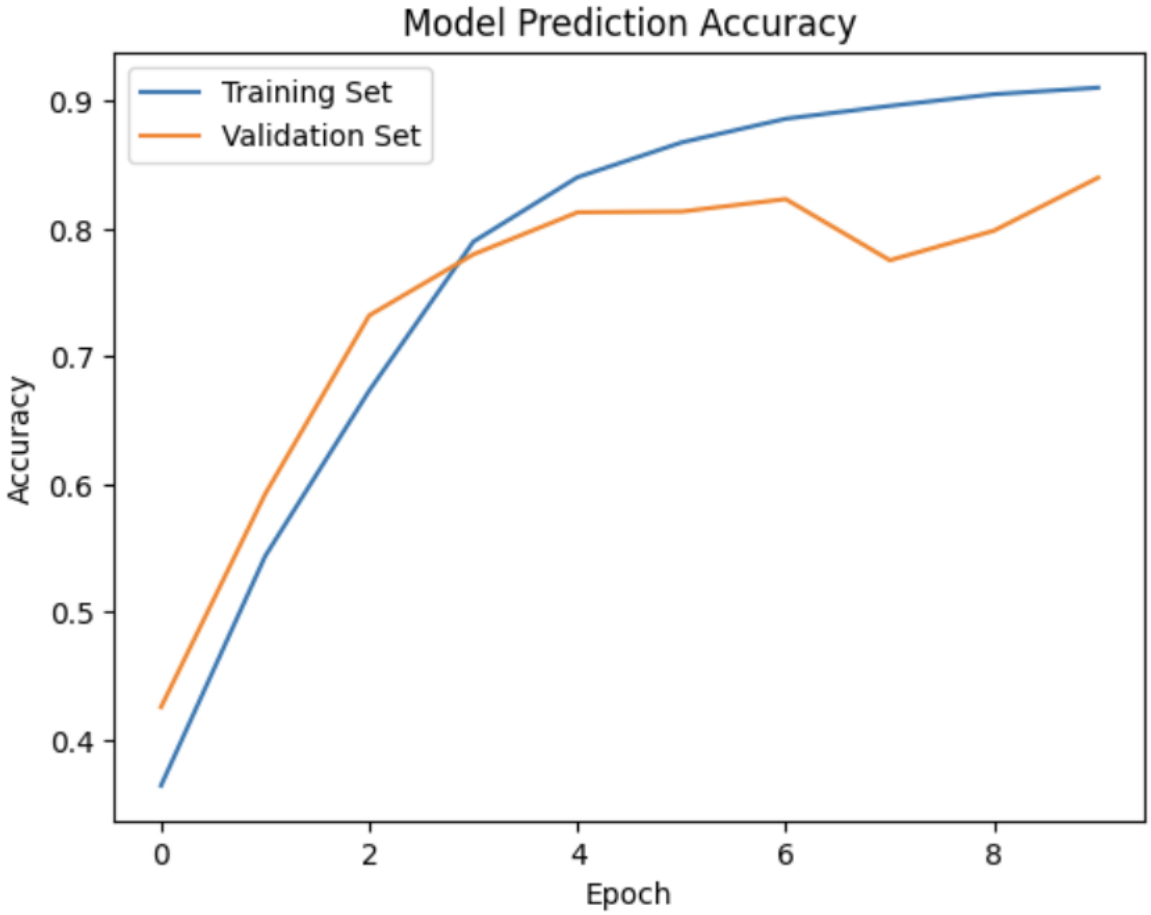
Additionally, after these Convolutional Layers, my model additionally has 3 Linear or Dense Layers. These are more standard neural network layers, without any convolution being applied to them. In my model, these have 128, 64, and 32 layers. These also use Leaky ReLU as the activation layer and have dropout layers to make the model more robust. Finally, we make it to the output layer, which has 3 nodes, 1 for each of the 3 possible outcomes. The activation layer for this final layer is a Softmax

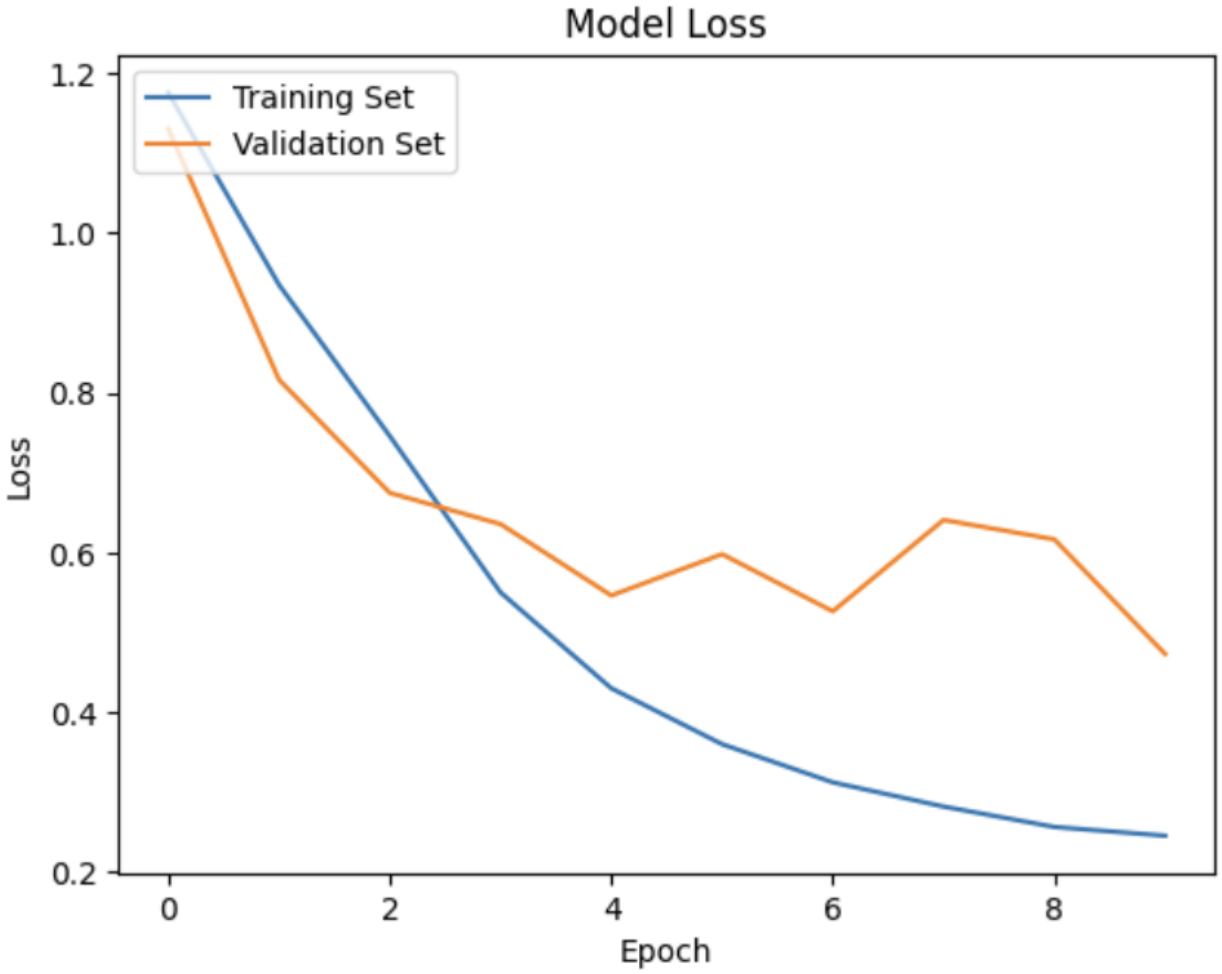
layer, a layer used in multiclass classification problems in order to have our network create probabilities for each of our classes [9].

Finally, before we actually begin training our model, we need to decide on a loss function and optimizer. Loss functions are what we use to benchmark our neural network, and so this can have a huge effect on our model. The loss function used in my model is a Sparse Cross Entropy Loss function. This loss function works by penalizing our prediction based on how far it is from the actual outcome. This penalty is logarithmic, so it's larger for large differences, trending towards 1, and smaller for small differences, trending towards 0 [10].

Let's now go over our optimizer, Adam. Adam is a combination of 3 different advances in optimization. The first Stochastic Gradient Descent, is a variant of the standard Gradient Descent algorithm, optimizing based on only a subset of the data, rather than the entire dataset. This allows us to get faster results, but also means that we have a much less consistent move towards our optima. The next addition, RMSProp, updates the learning rate of the optimizer based on a moving average of the size of recent gradients, helping the optimizer work on noisier datasets. Lastly, Momentum updates our parameters themselves (the weights and biases of our nodes) based on a moving average of our gradients, which helps our optimization algorithm move faster towards our optima. Adam combines all 3 of these developments together into one optimization package [11].

So now that we've gone over the different aspects of this custom model, how does it perform? This custom model manages to get a 91% successful prediction rate on my test data, which I would say is a great success. Additionally, it managed to converge very quickly, taking only 4 minutes to go through 10 epochs of processing (more specifically, 235.7 seconds). However, looking at how the model improved over time, the improvement on our validation loss and accuracy is quite erratic. This shows that, despite having a good prediction rate on our test data, the model may be prone to overfitting, and might give inconsistent results if we were to run it multiple times.

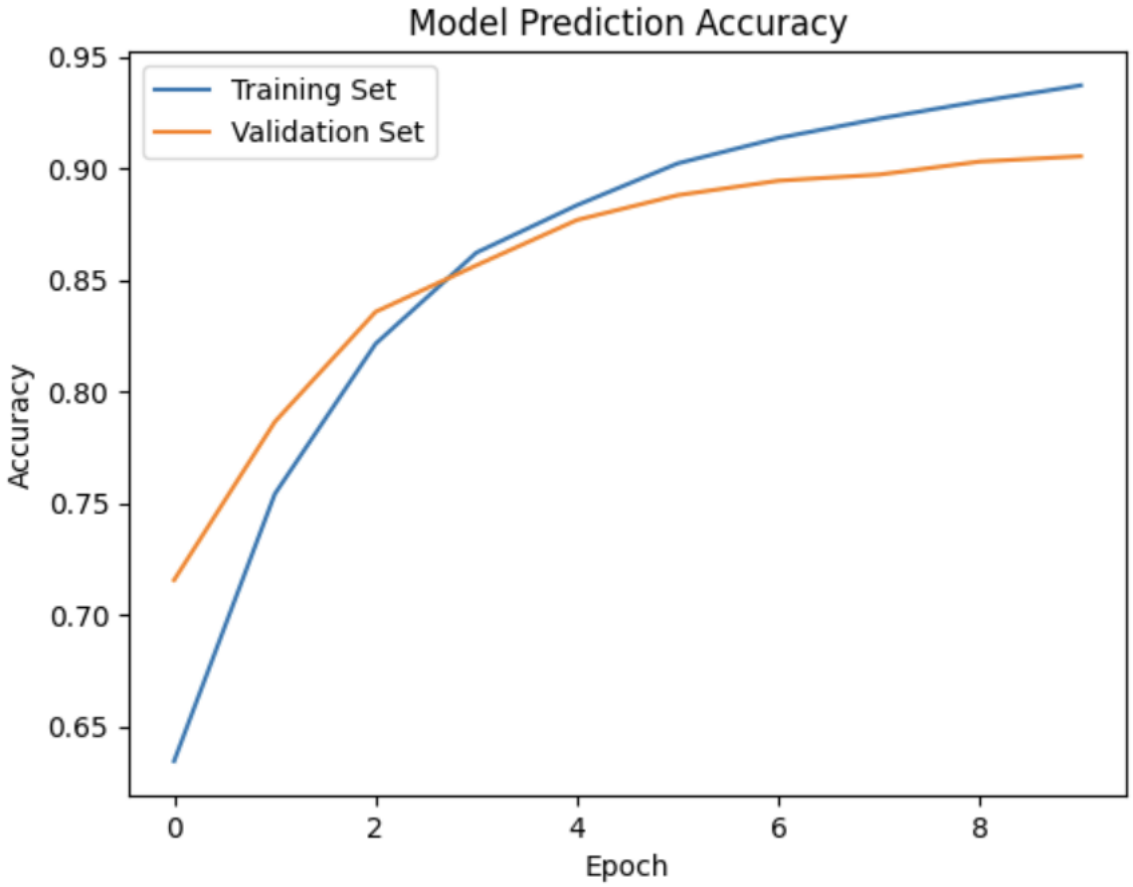

Model Prediction Accuracy
Training Set
Validation Set
Accuracy
Epoch
0.9
0.8
0.7
0.6
0.5
0.4
0
2
4
6
8

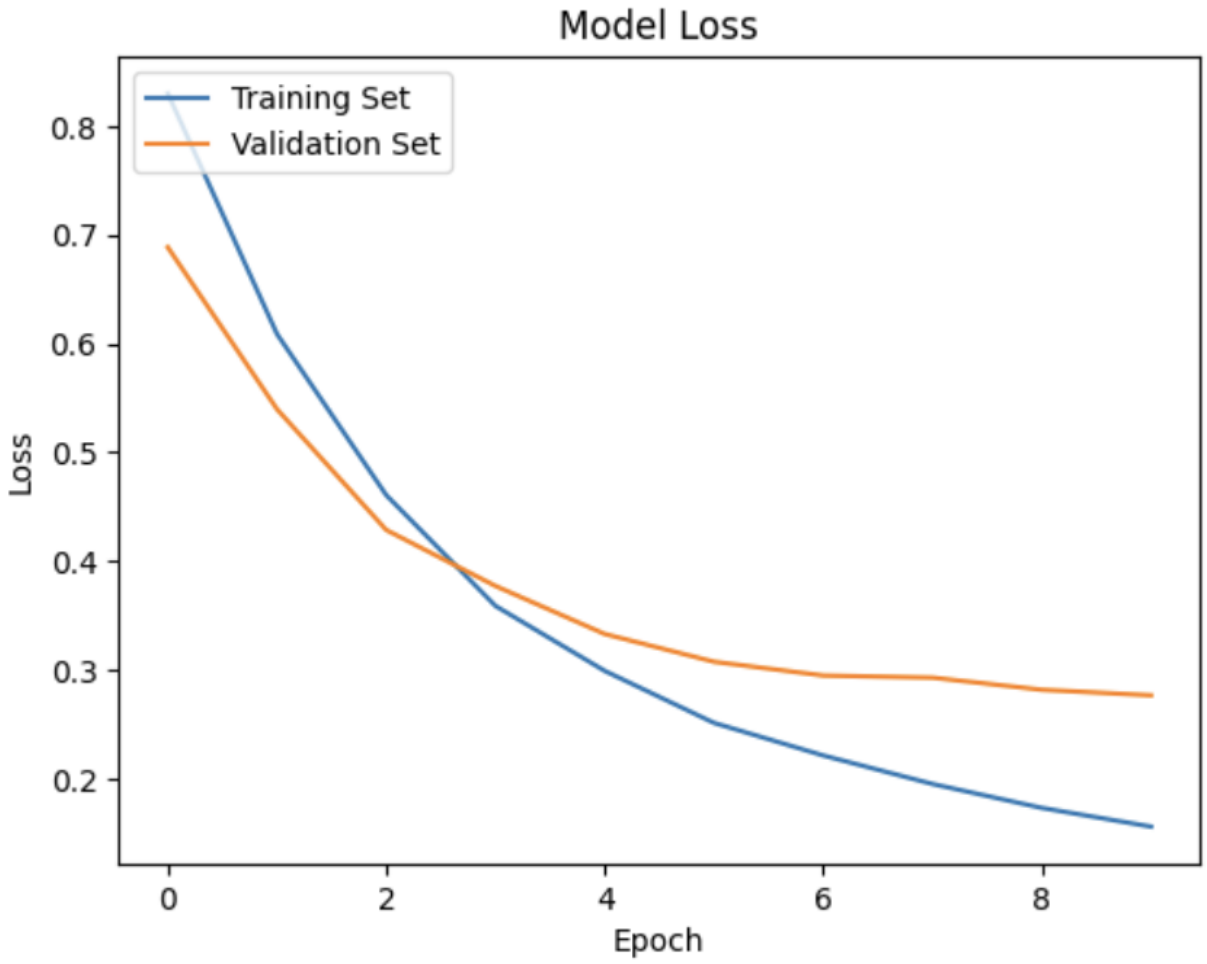


Now let's compare this custom model's results to that of the pre-built BERT model provided by the Keras-NLP library. BERT stands for Bidirectional Encoder Representations from Transformers, works very differently than the convolutional neural network we looked at before. As its name suggests, BERT is a Transformer model, which is a model that foregoes Convolutions and Recurrent Neural Networks and focuses entirely on attention mechanisms. Attention mechanisms allow the model to take into account all hidden states when making predictions, rather than just the input of the previous state. The Bidirectional aspect of the model comes from the model masking several of the inputs of the model, and attempting to additionally train the model on predicting these hidden inputs [12].

But how does the BERT Model compare to our custom CNN? An important note before continuing is that the BERT Model does its own model pre-processing, so I submitted my results as pure strings, rather than submitting the processed data I used for my custom model. With this in mind, our results show that the Bert Model has a slightly higher accuracy rate, correctly predicting 95% of our test set. Additionally, our

model improved much more smoothly than our custom model, which could have much more erratic results on how our validation loss and accuracy were improving over time. However, the training time was significantly slower than our custom model, taking over an hour and a half to go through 10 epochs of our data (more specifically, it took 5763.5 seconds). This means that, if you're working on data where accuracy is incredibly important, a BERT model might be a better choice, but if you need results quickly, a standard CNN can provide comparable results in a fraction of the time.





How would I potentially go about improving these results in the future? First off, my training, validation, and test data all come from the same source, which might not be representative of all Twitter posts. Therefore, these models could have worse results when predicting random tweets outside of this dataset. Gathering additional data would help shore up this possibility. Creating a web portal to allow users to enter their own tweets, get a sentiment prediction, and input what the correct response would've been might be a good way to both show off the model and gain additional data to improve the results. Additionally, the dataset has support for Entity-Level Sentiment Analysis, which would allow for more fine-grained sentiment analysis about individual topics. An Entity-Level model could have better results when analyzing sentiments towards specific individuals or things, as you'd be able to have different words have different results in how they affect the sentiment of a tweet based on the topic.

Citations:

[1]: Person, and Sheila Dang. “Exclusive: Twitter Is Losing Its Most Active Users, Internal Documents Show.” *Reuters*, Thomson Reuters, 26 Oct. 2022, https://www.reuters.com/technology/exclusive-where-did-tweeters-go-twitter-is-losing-its-most-active-users-internal-2022-10-25/.

[2]: Manning, Christopher D., et al. “Stemming and Lemmatization.” *Introduction to Information Retrieval*, Cambridge University Press, New York, 2019.

[3]: Brownlee, Jason. “How to Prepare Text Data for Deep Learning with Keras.” *MachineLearningMastery.com*, 7 Aug. 2019, https://machinelearningmastery.com/prepare-text-data-deep-learning-keras/.

[4]: “Artificial Neural Networks and Its Applications.” *GeeksforGeeks*, GeeksforGeeks, 11 Apr. 2023, https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/#.

[5]: “Convolutional Neural Network.” *DeepAI*, 17 May 2019, https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network.

[6]: “Activation Functions.” *Activation Functions - ML Glossary Documentation*, https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html.

[7]: DeepAI. “Batch Normalization.” *DeepAI*, DeepAI, 17 May 2019, https://deepai.org/machine-learning-glossary-and-terms/batch-normalization.

[8]: Brownlee, Jason. “A Gentle Introduction to Dropout for Regularizing Deep Neural Networks.” *MachineLearningMastery.com*, 6 Aug. 2019, https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/.

[9]: “Multi-Class Neural Networks: Softmax | Machine Learning | Google Developers.” *Google*, Google, https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax.

[10]: Koech, Kiprono Elijah. “Cross-Entropy Loss Function.” *Medium*, Towards Data Science, 16 July 2022, https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e.

[11]: Alabdullatef, Layan. “Complete Guide to Adam Optimization.” *Medium*, Towards Data Science, 2 Sept. 2020, https://towardsdatascience.com/complete-guide-to-adam-optimization-1e5f29532c3d.

[12]: Kulshrestha, Ria. “Keeping up with the Berts.” *Medium*, Towards Data Science, 22 Nov. 2020, https://towardsdatascience.com/keeping-up-with-the-berts-5b7beb92766.

[13]: Koehrsen, Will. “Neural Network Embeddings Explained.” *Medium*, Towards Data Science, 2 Oct. 2018, https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526.

[14]: Passionate-Nlp. “Twitter Sentiment Analysis.” *Kaggle*, 9 Aug. 2021, https://www.kaggle.com/datasets/jp797498e/twitter-entity-sentiment-analysis.

[15]: Mwiti, Derrick. “NLP Essential Guide: Convolutional Neural Network for Sentence Classification.” *Cnvrg*, 16 June 2021, https://cnvrg.io/cnn-sentence-classification/.