Machine Learning Assignment

(1)

a) Decision trees are grown deep, meaning they often lead to overfitting of the data they are trained on. [1] Thus, decision trees perform well when predicting outcomes from training datasets but perform significantly worse on test data. This is, because the decision tree does not only capture the underlying relations of predictors and outcome variables, but also the noise of the training data. Thus, decision trees suffer from high variance, but have low bias.

This is where bagging comes in. Bagging first utilizes bootstrapping, i.e., repeated sampling from the original training data with replacement, where for each sample, we fit a decision tree. In specific, bootstrapping involves the creation of B bootstrap datasets of the training data, where each data set includes a different subset of the original training data with the same size as the original training data set. After fitting decision trees on B bootstrap datasets, we have B individual decision trees with high variance, but low bias.

This is where the second step of bagging joins. For a given set of independent observations $Z_1, \dots, Z_B$, each with variance σ^2, the variance of the average of all observations $\boldsymbol{Z}$ is $\sigma^2/_B$. Bagging implements this procedure by taking many bootstrap samples, fitting a separate decision tree for each, and averaging the resulting predictions. Thus, we predict $f^1(x), \dots, f^B(x)$ and average those predictions into a single low-variance low-bias regression tree with the following prediction:

$$f^{ave}(x) = \frac{1}{B}\sum_{b=1}^{B} f^b(x)$$

Classification trees are conceptually similar, just they take a majority vote.[2] For the above to hold, however, each tree must represent an independent observation of the bootstrap datasets. This assumption will not always be satisfied.

b) Random forests provide an improvement over bagged decision trees by ensuring independent observations for each bootstrapped decision tree. Similar to bagging, random forests involve two steps. First, they build on a number of decision trees on bootstrapped samples. Second, they average B bootstrapped decision trees to arrive at a low-bias low-variance decision tree. The crucial difference being that while fitting decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. Bagging utilizes all predictors for each bootstrapped decision tree, i.e., $m = p$, while random forests restrict the number of predictors to typically $m \approx \sqrt{p}$ candidates.[3]

Suppose when fitting a decision tree there exist one overarchingly strong predictor, alongside some remaining predictors with lower predictive strength. During bagging, most of the fitted decision trees will use the strong predictor in the top split. Hence, bagged trees will result in similar structures and when averaging all bagged trees, the individual trees are not independent. If they are correlated, it

[1] Nagpal, A. 2017. Decision Tree Ensembles – Bagging and Boosting. Accesses on 10 December 2020. <https://towardsdatascience.com/decision-tree-ensembles-bagging-and-boosting-266a8ba60fd9>

[2] Steorts, R. C. Tree Based Methods: Bagging, Boosting, and Regression Trees. Accessed on 10 December 2020. <http://www2.stat.duke.edu/~rcs46/lectures_2017/08-trees/08-tree-advanced.pdf>

[3] Ibid.

lessens the reduction in variance, the goal of bagging. With random forests, on average $(p-m)/p$ of the top splits are forced not to consider the strong predictor. Thus, fitting decision trees over $m < p$ predictors will yield more variety in decision trees and lowers between-tree-correlation. Hence, averaging over random forests will generate larger reduction in variance and more reliable predictions.

c) Table 1 (in Jupyter Notebook) represents the confusion matrix from (1) c). It contains 286 predictions for whether a student dropped out or not, with a Misclassification Rate (MR) of 0.056, meaning roughly 5.6% of the predictions were wrong, i.e., they correspond to a Type I or Type II error and classify True Negatives as Predicted Positives or True Positives as Predicted Negatives respectively.[4]

Further, the classifier in Table 1 has a True Positive Rate and False Positive Rate of $TPR = 0.921$ and $FPR = 0.048$, meaning that 92.1% of the True Positives are correctly classified as Predicted Positives and 4.8% of the True Negatives are wrongly classified as Predicted Positives. Additionally, the classifier has a Null Error Rate (NER) of 0.734. The NER represents the rate by which a classifier would be wrong if it always predicted the majority class. A large difference between the NER and the MR as well as high TPR and low FPR indicate a good predictor.

In Figure 1, the classifier of Table 1 corresponds to a ROC curve with an area under the curve of close to 1, where if $AUC = 1$ the classifier would predict all values correctly. Point A on the ROC curve corresponds to the classifier in Table 1 with a threshold p^* around 0.5, since it is around halfway down the ROC curve, minimizing both TPR and FPR.

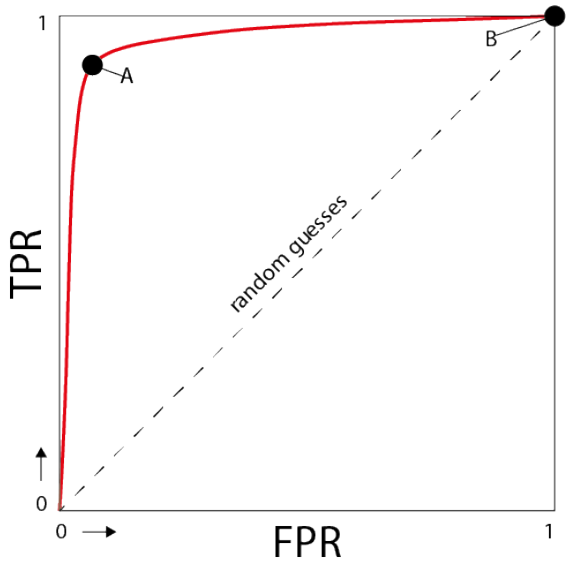

Figure 1: ROC curve

Important for the results above to hold is that the confusion matrix is relatively balanced, meaning that there are a considerable number of True Positives and Negatives as well and Predicted Positives and Negatives. This is the case. As a result, Table 1 represents good evidence that the questionnaire can predict college dropout when considering a subset of 20 predictors.

[4] Simple Guide to Confusion Matrix Terminology. Data School. Accessed on 10 December 2020. <https://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/>

(2)

b) Lasso regression is a regularized regression that leads to dimensionality reduction. The lasso regression penalizes the least-squares cost function $\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2$ via a $l1$-regularizer $\alpha \sum_{i=1}^{m}|\beta_j|$, where α determines the regularization strength, i.e., the amount of shrinkage. A higher α shrinks the coefficients of the lasso regression, and vice versa, where $\alpha = 0$ corresponds to a standard multiple linear regression (MLR) and $\alpha = \infty$ corresponds to a model without any coefficients. Hence, depending on α, multiple coefficients can collapse to zero, which determines the dimensionality reduction.

Principal Components Regression (PCR) is another dimensionality reducing regularized regression. PCR is a MLR with a reduced range of coefficients per Principal Component Analysis (PCA). In MLR, we have two matrices representing the raw data and the dependent variable (see Figure 2). The MLR can be written as $\boldsymbol{y} = \boldsymbol{Xb}$, where everything is a vector. The coefficient vector $\boldsymbol{b}$ is solved by equating $\boldsymbol{b} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y}$. The PCR reduces $\boldsymbol{X}$ into $\boldsymbol{T} = \boldsymbol{XP}$ through a PCA and the MLR can be written as $\boldsymbol{y} = \boldsymbol{Tb}$ and $\boldsymbol{b} = (\boldsymbol{T}'\boldsymbol{T})^{-1}\boldsymbol{T}'\boldsymbol{y}$.[5]

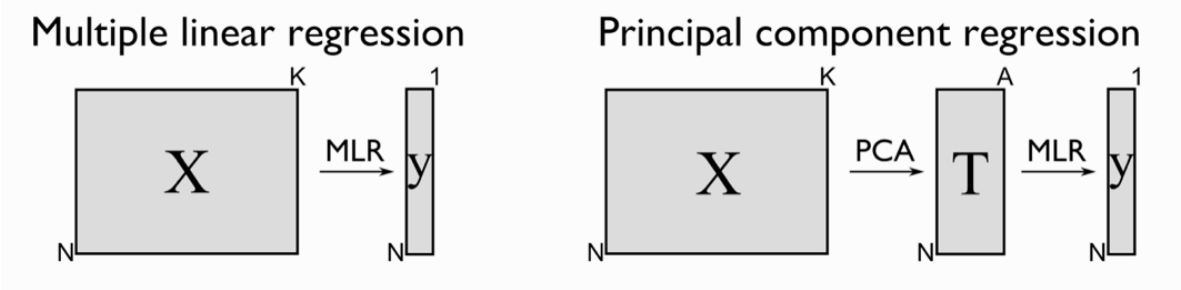

Figure 2: MLR and PCR[6]

Now, within the given task the predictors $\boldsymbol{X}$ are correlated, increasing the variance of any MLR trying to predict the outcome $\boldsymbol{y}$. To investigate which of the above techniques results in a lower MSE, consider that both are techniques for dealing with multi-collinearities in the predictor space of a regression, the main difference being that the dimensionality reduction within PCR happens before the MLR, while it happens simultaneously for the lasso regression. Both techniques trade off a degree of bias to the regression estimates by removing some explanatory variables, to reduce collinearity.

The main disadvantage of the PCR is it does not consider the response variable during its PCA. Instead, it only focuses on the magnitude of the variance of each component. However, there is no reason for the principal components with the highest variance to simultaneously have the highest predictive power.[7] The main disadvantage of the lasso regression is that to arrive at a dimensionality reduction of only two predictors, the shrinkage factor is so high that the remaining predictors are likely to be shrunken as well, further increasing the bias of the remaining coefficients. Figure 3 shows that the remaining two coefficients are likely far from their unconstrained values.

[5] Principal Component Regression (PCR). 2020. Process Improvement Using Data. Accessed 13 December 2020. <https://learnche.org/pid/latent-variable-modelling/principal-components-regression>

[6] Ibid.

[7] Jolliffe, I. 1982. A Note on the Use of Principal Components in Regression. Journal of the Royal Statistical Society. Series C (Applied Statistics), 31(3), 300-303. Accessed 16 December 2020. < www.jstor.org/stable/2348005>

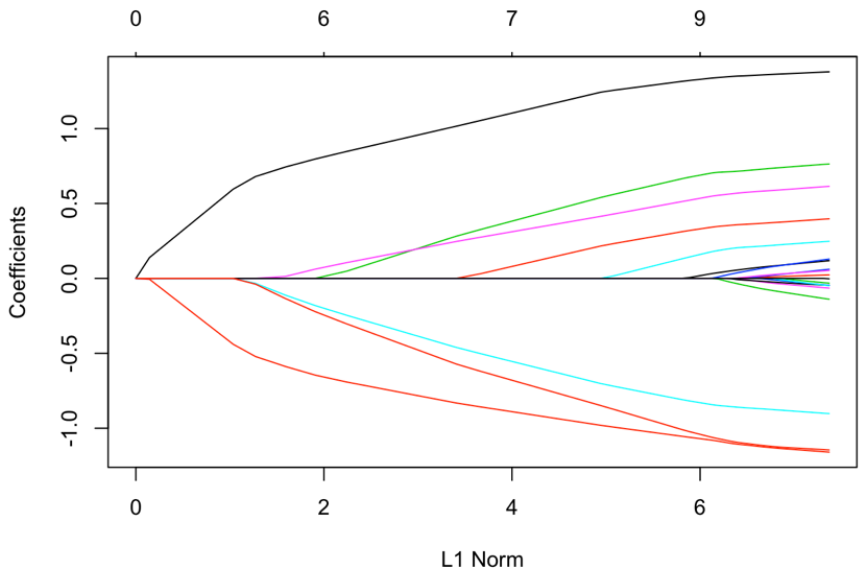

Figure 3: Lasso regression coefficients

In general, for a small number of dimensions considered, the PCR tends to outperform the lasso regression if the first few principal components explain most of the variation in the predictors.[8] In the given task, the number of dimensions considered is very small, but variance explained by the first two principal components is only around 40%. As a result, it is unclear which technique would result in a MSE when testing the trained models on novel test data.

[8] Kelly, R. 2014. Linear Model Selection & Regularization. Accessed 9 December 2020. <https://rstudio-pubs-static.s3.amazonaws.com/22067_48fad02fb1a944e9a8fb1d56c55119ef.html#overview-and-definitions>

a) The decision boundary of a SVM can be formulated as $f(\boldsymbol{x}) = \beta_0 + \sum_i \alpha_i K(\boldsymbol{v_1}, \boldsymbol{v_2})$. When applying the polynomial kernel with degree $d = 2$, we get $f(\boldsymbol{x}) = \beta_0 + \sum_i \alpha_i K\left(1 + \boldsymbol{v_1^T}\, \boldsymbol{v_2}\right)^2$. Now, since $\boldsymbol{v_i} = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$, we get $\boldsymbol{v_1^T}\, \boldsymbol{v_2} = x_1 x_2 + y_1 y_2$ and $K\left(1 + \boldsymbol{v_1^T}\, \boldsymbol{v_2}\right)^2 = (x_1 x_2 + y_1 y_2 + 1)^2 = \boldsymbol{x}^2 + \boldsymbol{y}^2 + 2\boldsymbol{x^T y} + 2\boldsymbol{x} + 2\boldsymbol{y} + 1$, where $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and $\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$. Thus, overall, we get $f(\boldsymbol{x}) = \beta_0 + \sum_i \alpha_i K(\boldsymbol{v_1}, \boldsymbol{v_2}) = f(\boldsymbol{x}) = \beta_0 + \sum_i \alpha_i (\boldsymbol{x}^2 + \boldsymbol{y}^2 + 2\boldsymbol{xy} + 2\boldsymbol{x} + 2\boldsymbol{y} + 1)$, which is a decision boundary as a conic section.[9]

b) Support Vector Machines (SVM) (with a linear kernel being Support Vector Classifiers (SVC)), construct a hyperplane to classify data into subgroups. The simplest SVM is a Maximum Margin Classifier (MMC), maximizing the margin of the hyperplane such that no data points lie within that margin, where the margin is the distance between the hyperplane and the so-called support vectors. Thus, the resulting hyperplane depends only on close data points to the hyperplane. The MMC works only for linearly separable data, because the hyperplane can only be constructed if $y_i f(x_i) > 0$ for all i, where y_i represents the outcome classification (either 1 or -1) and $f(x_i)$ represents the distance of the data point from the hyperplane, which can take positive values if the datapoint is above the hyperplane and negative values if it is below the hyperplane. The margin in the MMC is maximized according to

$$\max_{\boldsymbol{\beta}} M, \text{ subject to } \|\boldsymbol{\beta}\|^2 = 1 \text{ and } y_i f(x_i) \geq M,$$

where $\|\boldsymbol{\beta}\|^2 = 1$ is the normalization of the orthogonal vector to the hyperplane and M is the margin. It can be seen that if a point that is classified as $y_i = 1$, but lies below the hyperplane that the constraint is not fulfilled, and no margins can be constructed. The SVC (and SVM) instead allow for misclassification. The adapted optimization problem is:

$$\max_{\boldsymbol{\beta}} M, \text{ subject to } \|\boldsymbol{\beta}\|^2 = 1 \text{ and}$$
$$y_i f(x_i) \geq M(1 - \varepsilon_i), \text{ where } \varepsilon_i \geq 0 \text{ and } \textstyle\sum_i \varepsilon_i \leq B.$$

We note a slack variable ε_i added to the margin constraint, which is constrained to ≥ 0 and sum to no more than the budget B. If ε_i increases, $M(1 - \varepsilon_i)$ decreases for a given M, or stays constant for a larger M. The larger the B, the larger the ε_i, meaning that M increases. Thus, B is a regularization parameter and can be tuned to minimize the test error of a model. When B is small, the optimization problem allows a small amount of misclassification, and the SVC will have low bias but does not generalize well because of high variance, i.e., we run the risk of overfitting. On the other hand, if B is large, the amount of misclassification allowed increases, and the resulting SVC runs the risk of having a large bias. If $B = 0$, SVC = MMC.

SVC performs badly if the data points are non-linearly distributed, which is why the kernel trick has been invented. Common kernels include the polynomial kernel $k(x, x') = (1 + x^T x')^d$, and the radial basis kernel $k(x, x') = exp\left(-{\|x - x'\|^2}/{2\sigma^2}\right)$ alongside the linear kernel $k(x, x') = x^T x'$ discussed above. The polynomial and radial basis kernel map the data into higher-dimensional space where the classification is solved linearly. Thus, the kernel trick allows classifiers to look beyond a given set of input features — at combinations of them. The polynomial kernel is of degree-d, where d defines the

[9] Support Vector Machines. Accessed 12 December 2020. <https://ai6034.mit.edu/wiki/images/SVM_and_Boosting.pdf>

dimensionality of the kernel. The dimensionality determines the flexibility of the classifier, where if dimensionality increases, bias decreases, but variance increases. The radial basis kernel is infinitely dimensional, and its hyper-parameter is represented by σ, where $1/\sigma$ reflects the radius of influence of data points as support vectors on the classifier. Thus, low σ can constrain the resulting model too much and hence do not capture the complexity of the data points involved. Large σ can lead to a model that includes only the support vectors of the classifier itself. The optimal σ lies in the middle.

In general, a linear kernel performs well for linear functions, because it is parametric and does not grow in complexity as the size of the data set grows. Having said that, the linear kernel performs badly for non-linear functions. A quadratic kernel is parametric too, whereas the radial-basis kernel is nonparametric, meaning it grows in complexity with the size of the dataset. The radial-basis kernel is able to present more and more complex relationships as the dataset grows while the quadratic kernel's size of complexity is fixed by its degree. Hence, if the dataset involves a non-linear function and we can only make weak assumptions about it, the radial-basis kernel will present more information than the quadratic kernel. However, if the dataset involves a non-linear function and strong assumptions about the relationship in the data, then a polynomial kernel performs better due to decreased complexity.

(4)

a) The following is the derivation of the backpropagation algorithm for a neural network with one hidden layer. The backpropagation algorithm learns and updates the weights

$$\Delta\, w_{ji} = -\alpha \frac{\partial L}{\partial w_{ji}}$$

of a neural network according to some learning rate α and some loss function L.

$$L = \frac{1}{2}\sum_{k\in O}(t_k - a_k)^2$$

The derivation will utilize the following terminology, where x_{ij} is the ith input to unit j, w_{ji} is the weight of ith input to unit j, $z_j = \sum_i w_{ji}x_{ji}$, and $a_j = g(z_j)$, where g is an activation function. O is the set of output units and t_k is the target value associate with the kth output. The aim is to derive $\frac{\partial L}{\partial w_{ji}}$, where $\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial w_{ji}}$, because the weight w_{ji} can influence the neural network only through the weighted sum of input z_i. In general, there are two separate derivations, one for the output units O, and one for the internal units h.

The derivation of $\frac{\partial L}{\partial w_{ji}}$ for the output unit is the following:

$$\frac{\partial L}{\partial w_{kh}} = \frac{\partial L}{\partial z_k}\frac{\partial z_k}{\partial w_{kh}}$$

Taking it step by step, the derivation starts with the following

$$\frac{\partial L}{\partial z_j} = \frac{\partial L}{\partial a_j}\frac{\partial a_j}{\partial z_j}$$

because z_j can influence the network only through a_j, where $\frac{\partial a_i}{\partial z_i}$ is simply $g'(z_j)$. Next, we derive $\frac{\partial L}{\partial a_j}$

$$\frac{\partial L}{\partial a_j} = \frac{\partial}{\partial a_j}L = \frac{\partial}{\partial a_j}\frac{1}{2}\sum_{k\in O}(t_j - a_j)^2$$

where $\frac{\partial}{\partial a_j}\frac{1}{2}\sum_{k\in O}(t_j - a_j)^2 = \frac{\partial}{\partial a_j}\frac{1}{2}(t_j - a_j)^2 = -(t_j - a_j)$, because the derivatives $\frac{\partial}{\partial a_j}\frac{1}{2}(t_j - a_j)^2$ will be zero for all output units k except for the case when $k = j$. Hence, $\frac{\partial L}{\partial z_j} = \frac{\partial L}{\partial a_j}\frac{\partial a_j}{\partial z_j} = -(t_j - a_j)g'(z_j)$, which we denote as δ_j. Putting things together, we get

$$\frac{\partial L}{\partial w_{kh}} = \frac{\partial L}{\partial z_k}\frac{\partial z_k}{\partial w_{kh}} = \delta_k a_h$$

and

$$\Delta w_{kh} = -\alpha \frac{\partial L}{\partial w_{kh}} = -\alpha \delta_k a_h$$

which describes the change in the weight of the hidden input h on output layer k.

The derivation for an internal unit is conceptually similar, only that there are no target values directly available to indicate the loss.[10] Instead, we consider every unit downstream of unit j in the neural network, because Δw_{ji} impacts the network outputs only through downstream units. In the particular case of a two-layer neural network, this means that

$$\frac{\partial L}{\partial w_{hi}} = \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial z_j} \frac{\partial z_j}{\partial w_{hi}}$$

where

$$\frac{\partial L}{\partial z_j} = \sum_{k \in ds(j)} \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial z_j} = \sum_{k \in ds(j)} \delta_k \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial z_j} = \sum_{k \in ds(j)} \delta_k w_{kj} g'(z_j)$$

and $\frac{\partial L}{\partial z_k} = \delta_k$ from above and $\frac{\partial z_k}{\partial z_j} = w_{kj} g'(z_j)$, because two layers are connected by a weight and an activation function. We denote $\frac{\partial L}{\partial z_j} = \delta_j$. Putting everything together, we get

$$\frac{\partial L}{\partial w_{hi}} = \frac{\partial L}{\partial z_h} \frac{\partial z_h}{\partial w_{hi}} = \sum_{k \in ds(h)} \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial z_j} x_i = g'(z_j) \sum_{k \in ds(h)} \delta_k w_{kh} x_i = \delta_h x_i$$

and

$$\Delta w_{hi} = -\alpha \frac{\partial L}{\partial w_{hi}} = -\alpha \delta_h x_i$$

which describes the change in the weight of the input i on output layer k via hidden layer h.

d) The research paper describes a novel sequence transduction model — a model that reasons from specific (training) cases to specific (test) cases, such as Machine Translation algorithms (which is the focus of the paper). The model is called Transformer, and it contrasts with previous models, such as the Long Short-Term Memory (LSTM) model, because it reasons via the content of its inputs instead of the positional values.

In general, Machine Translation needs some memory to translate a sentence. Machine Translation cannot classify any word in a sequence of words effectively if it does not understand the context of that specific word. Recurrent Neural Networks (RNNs) address the issue by forming a loop.

[10] Liskowski, P. Derivation of Backpropagation Algorithm for Feedforward Neural Networks. Accessed 14 December 2020. <http://www.cs.put.poznan.pl/pliskowski/pub/teaching/eio/lab1/eio-supplementary.pdf>

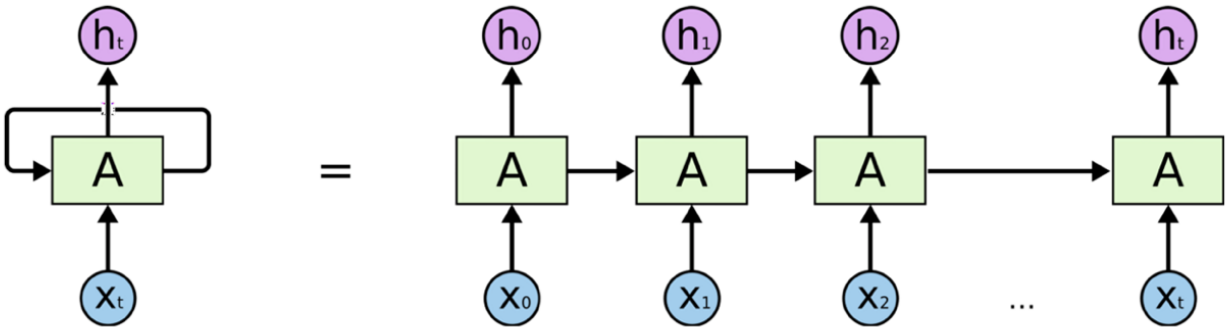

Figure 4: unrolled RNN[11]

In Figure 4, X_t represents the input (e.g., English word), A represents a transformation of an encoder/decoder, where encoders abstract each input into one whole sequence to be passed on to the decoder, which translates the sequence bit by bit into h_t outputs. The abstracted word sequence passed on by the encoder allows the decoder to connect information across inputs although it is working through each word sequentially. This works well when the word sequence is short, and information relevant to the current input of the decoder relatively recent, i.e., only a few loops in the past. Unfortunately, RNNs perform badly when it comes to long-term dependencies, where the relevant information necessary to translate an input is far in the past. Theoretically, the RNN could retrieve the information about long-term dependencies, but every time an input is transformed by a decoder, the entire information given by the abstraction of the encoder is transformed, meaning that dependencies vanish over time.[12] LSTMs have been developed (as a special kind of RNN) to deal with the problem of long-term dependencies.[13] LSTMs feature an additional transformation of inputs in the decoder stage — a "cell state" which allows for linear interactions between all of the inputs up to the current cell, thus preserving information about prior inputs over time. Nevertheless, LSTMs perform badly over long word sequences, the reason being that although the "cell-state" preserves the information about the previous inputs, the prominence of word contexts far in the past decreases exponentially. Additionally, the LSTM cannot be parallelized.

In contrast, the Transformer does not utilize an RNN and references by content instead of position. The Transformer still deploys an encoder-decoder structure. Figure 5 outlines the architecture of the Transformer as presented by the research paper, with encoders on the left and decoders on the right.[14] I will go through the former first, and subsequently leave some comments about the difference between the encoder and decoder before finishing off with the practical implications.

[11] Giacaglia, G. 2019. How Transformers Work. Accessed 14 December 2020. <https://towardsdatascience.com/transformers-141e32e69591>

[12] Bengio, Y. Simard, P. Frasconi, P. 1994. Learning Long-Term Dependencies with gradient Descent is Difficult. IEEE Transactions on Neural Networks, 5(2). <http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf>

[13] Hochreiter, S. Schmidhuber, J. 1997. Long Short-Term Memory. Neural Computation 9(8):1735-1780. <http://www.bioinf.jku.at/publications/older/2604.pdf>

[14] Vaswani, A. Shazeer, N, Parmar, N. Uszkoreit, J. Jones, L. Gomez, A N. Kaiser, L. Polosukhin, I. 2017. Attention is All You Need. <https://arxiv.org/pdf/1706.03762.pdf>

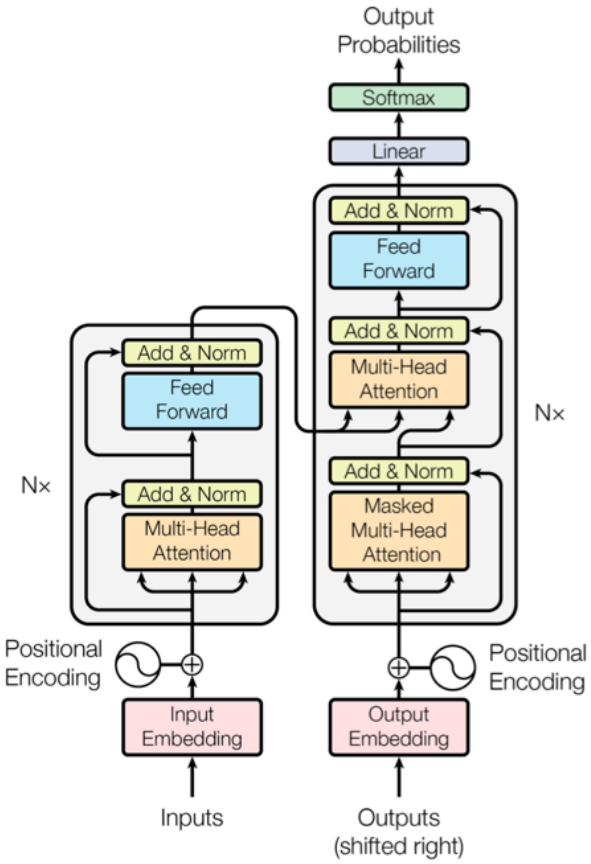

Figure 5: Transformer model architecture[15]

The encoder consists of two main layers, attention layer and a feed-forward neural network. The attention layer consists of six steps. The resulting output is a matrix containing attention values about the relation of words within a sentence that is fed into a feed-forward neural network consisting of two linear transformations before being passed onto the next attention layer, of which there are six in total. Words have higher relations to each other if the attention value is higher, indicated in Figure 6. The attention value is a measure about how likely a word is related to another word in the word sequence analyzed. In Figure 6 the Transformer could just as well have associated the “es” with the street instead of the animal, but because it learned that being tired is likely to be an attribute of an animal rather than a street, it associates it with the animal in the sentence, which is further in the past in terms of the word sequence than the street. In other words, the Transformer is likely to pass a Winograd schema challenge because of reference by content whereas LSTMs lacks that differentiation.

[15] Ibid.

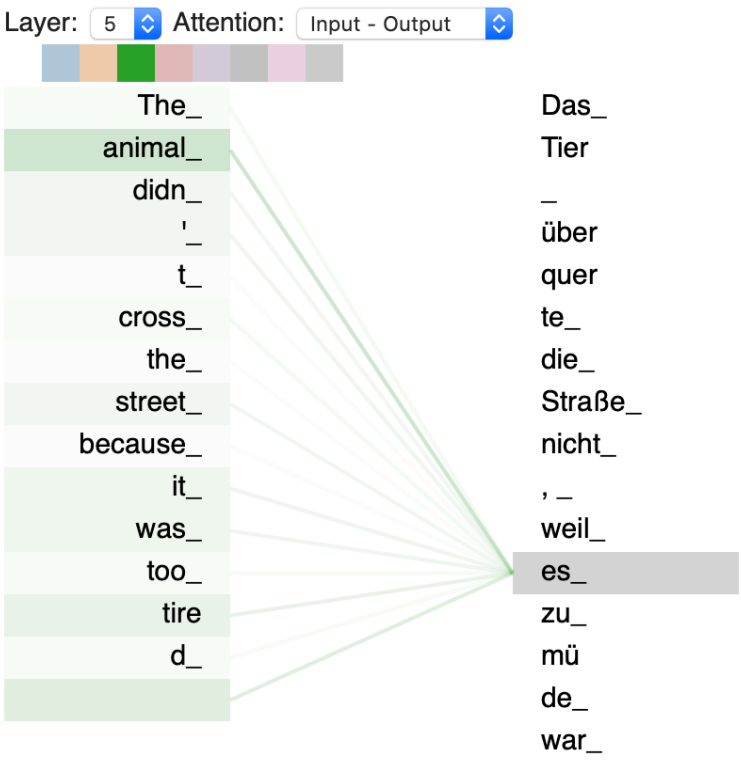

Figure 6: Transformer during Machine Translation[16]

Further, the Transformer utilizes what is called "Multi-Head Attention" (see Figure 7). This means that several attention layers are running in parallel. This enables the transformer to learn different features of the word sequence simultaneously. For example, one attention layer could be interested in the question of who the subject of "es" in Figure 6 is, while another parallel attention layer would investigate the question of who did what.

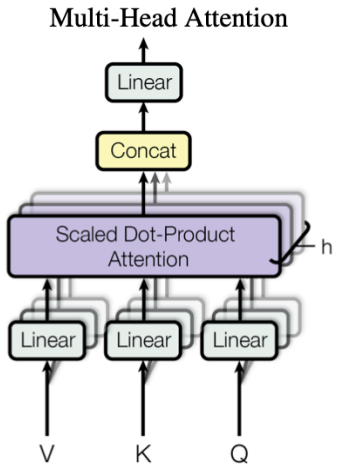

Figure 7: Transformer Multiheaded Attention[17]

[16] Tensor2Tensor Intro. Accessed 14 December 2020. <https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb#scrollTo=OJKU36QAfqOC>

[17] Vaswani, A. Shazeer, N, Parmar, N. Uszkoreit, J. Jones, L. Gomez, A N. Kaiser, L. Polosukhin, I. 2017. Attention is All You Need. <https://arxiv.org/pdf/1706.03762.pdf>

Since the attention layer references by content instead of positional value, the input has to be represented in order for the output to not simply be a bag of words. This is why the Transformer adds a positional encoder prior to the attention layer, which adds a vector to each input, marking it with a positional value. This completes the description of the encoder. The decoder is conceptually similar to the encoder, just that it adds a “Masked Multi-Head Attention”-layer which ensures that the predictions of a specific word in a sequence only relies on words prior to it. Finally, the predictions are linearly transformed and normalized via a “softmax” application so as to sum to one to be interpretable.

All in all, the Transformer presents a model of sequence transduction that is easier and faster to train, because it utilizes modern-day GPUs by relying on a large array of matrix multiplication and parallelization and offers more accurate results than previous sequential Machine Translation algorithms due to reference by content, effectively bridging long-term dependencies.