1. **The XOR Problem**: In the XOR problem, an neural network is asked to compute the output of the XOR function, that is (0,0) -> 0, (0,1) -> 1, (1,0) -> 1, (1,1) -> 0. This cannot be solved using a linear regression model since there is no linear rule seperating the two classes. This problem however can be solved by applying a transformation on the data such that it becomes linearly seperable. The question is weather a neural network can learn this transformation. The answer is yes, but it is a very complicated task for them and requires multiple hidden, non-linear layers (the problem is not linearly seperable and therefore cannot be solved with only a linear layer).
2. **Objective Function**: the objective function in neural networks is defined as how similar the true and the predicted values are. The objective function is proportional to the sum of the losses over the different labeled pairs. We are interested in finding the best parameters, so we would like to minimize the objective function with respect to the parameters.

$$\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}_\theta) \propto \sum_{i=1}^{n} \ell(\mathbf{y_i}, f_\theta(\mathbf{x_i})) \qquad \arg\min_\theta \mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}_\theta)$$

3. **Defining the Loss function**: there different loss meticas:
   a. 0-1 Loss: this loss function is ideal, especial where you cannot allow any false positive classifications (for example you don't want to notify a patient that he has cancer while this might be a false classification). However, this loss is not useful for neural networks since this function is not convex and therefore gradient based methods cannot be applied on it (the gradient is always zero).

$$\begin{cases} 0 & \mathbf{y} = \hat{\mathbf{y}} \\ 1 & \text{otherwise} \end{cases}$$

   b. Hinge Loss: The idea behind this loss is that the function doesn't care if the prediction is correct and gives it a loss of 0 (no matter how much more correct the prediction was) however if the prediction was incorrect, the function returns a penalty that is linearly proportional to how incorrect the prediction was. This loss is a differentiable and therefore can be used in neural networks.

$$t = \arg\max_i \mathbf{y}_{[i]}$$
$$p = \arg\max_{i \neq t} \hat{\mathbf{y}}_{[i]}$$
$$\ell_{\text{hinge}} = \max(0, 1 - (\hat{\mathbf{y}}_{[t]} - \hat{\mathbf{y}}_{[p]}))$$

Rewrite hinge loss in terms of w as $f(g(w))$ where $f(z) = \max(0, 1 - y\,z)$ and $g(w) = \mathbf{x} \cdot \mathbf{w}$

Using chain rule we get

$$\frac{\partial}{\partial w_i} f(g(w)) = \frac{\partial f}{\partial z} \frac{\partial g}{\partial w_i}$$

First derivative term is evaluated at $g(w) = x \cdot w$ becoming $-y$ when $\mathbf{x} \cdot w < 1$, and 0 when $\mathbf{x} \cdot w > 1$. Second derivative term becomes x_i. So in the end you get

$$\frac{\partial f(g(w))}{\partial w_i} = \begin{cases} -y\,x_i & \text{if } y\,\mathbf{x} \cdot \mathbf{w} < 1 \\ 0 & \text{if } y\,\mathbf{x} \cdot \mathbf{w} > 1 \end{cases}$$

   c. Log - Loss: (Cross Entropy) a very common loss function. It is derived from trying to understand how much information the prediction holds about the true label. This approach is probability based. In the case of hard loss, that is when there is only one correct label, the equation can be reduced. The function is defined over vectors of probabilities (in the rage of [0,1]) where each index indicates how probable this label is. The loss is proportional to the amount of uncertainty in the prediction, that is we get more penalty when the prediction is father away from 1. $\hat{y} = P(y = k|x)$.

$$\ell_{\text{cross-ent}} = -\sum_k \mathbf{y}_{[k]} \log \hat{\mathbf{y}}_{[k]}$$

for "hard" (0 or 1) labels: $\ell_{\text{cross-ent}} = -\log \hat{\mathbf{y}}_{[t]}$

4. Converting outputs to probabilities: when it comes to converting an output vector to probabilities, the softmax function is used. The softmax function maps values to the range of [0,1]. The normalization is crucial so the values will add up to 1.
    - the softmax function:

$$\text{softmax}(\mathbf{v})_{[i]} = \frac{e^{\mathbf{v}_{[i]}}}{\sum_{i'} e^{\mathbf{v}_{[i']}}}$$

Log-linear model
(aka "logistic regression")

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{x}\mathbf{W} + \mathbf{b})$$

$$\hat{\mathbf{y}}_{[i]} = \frac{e^{(\mathbf{x}\mathbf{W}+\mathbf{b})_{[i]}}}{\sum_i e^{(\mathbf{x}\mathbf{W}+\mathbf{b})_{[i]}}}$$

5. Gradient Decent: is an optimization algorithm that finds the minimum value of a function. The idea is to compute the loss of the function based on the whole data with respect to the current parameters, then find the gradients of the loss. The gradients are used to update the parameters for the next iteration. This process runs until a criterion is met and the resulting parameters are used as the optimal parameters for the task. This is a very slow approach since we are iterating over the whole data. The idea is usually visualized by thinking of a valley

Algorithm 1 Gradient Descent Training

Input:
- Function $f(\mathbf{x};\Theta)$ parameterized with parameters Θ.
- Training set of inputs $\mathbf{x_1},\ldots,\mathbf{x_n}$ and desired outputs $\mathbf{y_1},\ldots,\mathbf{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Compute the loss $\mathcal{L}(\Theta) = \sum_i L(f(\mathbf{x_i};\Theta),\mathbf{y_i})$
3: $\hat{\mathbf{g}} \leftarrow$ gradients of $\mathcal{L}(\Theta))$ w.r.t Θ
4: $\Theta \leftarrow \Theta - \eta_t\hat{\mathbf{g}}$
5: **return** Θ

with many trees. You want to get to the lowest point of the valley, but you cannot see it. The solution is to calculate the direction of descending (the gradient) with respect to where you are standing and follow that direction. After few steps, the process should be applied again. Notice that you can easily get stuck in a local minimum. This is the reason why initialization is important because string at different places (possibly at random, but same seed) can lead to different minimum points, that is for better optimization.

6. SGD: (Stochastic Gradient Decent) an improvement on the vanilla gradient decent. The idea is to compute the gradient on a single point and update the parameters based on that. This approach is very noise, since the gradient will be misleading in many cases. However it is much faster.

Algorithm 2 Online Stochastic Gradient Descent Training

Input:
- Function $f(\mathbf{x};\Theta)$ parameterized with parameters Θ.
- Training set of inputs $\mathbf{x_1},\ldots,\mathbf{x_n}$ and desired outputs $\mathbf{y_1},\ldots,\mathbf{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\mathbf{x_i},\mathbf{y_i}$
3: Compute the loss $L(f(\mathbf{x_i};\Theta),\mathbf{y_i})$
4: $\hat{\mathbf{g}} \leftarrow$ gradients of $L(f(\mathbf{x_i};\Theta),\mathbf{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t\hat{\mathbf{g}}$
6: **return** Θ

7. **Batched SGD**: is an improvement over SGD where instead of sampling only one point, we take a batch of points to compute the loos on. Slow than SGD but better performance. However in case we have GPU available, we will still take all points for the computation, because it is easy to parallelize. In addition, note that the loss is taken as the average loss cross the different sample points.

Algorithm 3 Minibatch Stochastic Gradient Descent Training

Input:
- Function $f(\mathbf{x};\Theta)$ parameterized with parameters Θ.
- Training set of inputs $\mathbf{x_1},\ldots,\mathbf{x_n}$ and desired outputs $\mathbf{y_1},\ldots,\mathbf{y_n}$.
- Loss function L.

while stopping criteria not met **do**
  Sample a minibatch of m examples $\{(\mathbf{x_1},\mathbf{y_1}),\ldots,(\mathbf{x_m},\mathbf{y_m})\}$
  $\hat{\mathbf{g}} \leftarrow 0$
  for $i = 1$ to m **do**
    Compute the loss $L(f(\mathbf{x_i};\Theta),\mathbf{y_i})$
    $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} +$ gradients of $\frac{1}{m}L(f(\mathbf{x_i};\Theta),\mathbf{y_i})$ w.r.t Θ
  $\Theta \leftarrow \Theta - \eta\hat{\mathbf{g}}$
return Θ

8. **Linear Classifier**: Linear classify is a simple network based on a linear operation. The input is multiplied by a weight matrix and added a bias term. The weights and the bias are tuned during the training process. In order to have multiple linear layers we need to break the linearity (otherwise we just get a really big single linear layer). This is done by passing the linear layer through a non-linear function.

$$f(\mathbf{x}) = \mathbf{W}\cdot\mathbf{x} + \mathbf{b}$$

9. **Non-Linear Classifier**: is a classifier that applies a non-linear function on the output of a layer, for example on a linear layer:

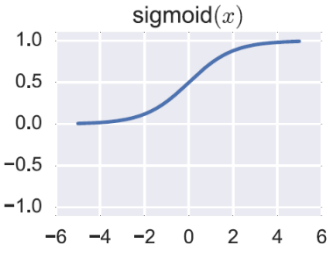
$$f_\theta(\mathbf{x}) = \mathbf{w}g(\mathbf{W}'\cdot\mathbf{x} + \mathbf{b}') + b$$

10. **MLP**: (multi-layet preceptron) is a stack of linear layers seperated by non linear activation functions. The parameters $\boldsymbol{\theta}$ that needs to be tuned are W^1, b^1, W^2, b^2 and W^3. This model is to solve very simple prediction problems.

$$\begin{aligned} f_\theta(\mathbf{x}) = \mathrm{NN}_{\mathrm{MLP2}}(\mathbf{x}) &= \mathbf{y} \\ \mathbf{h^1} &= g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1}) \\ \mathbf{h^2} &= g^2(\mathbf{h^1}\mathbf{W^2} + \mathbf{b^2}) \\ \mathbf{y} &= \mathbf{h^2}\mathbf{W^3} \end{aligned}$$

11. **Activation Functions**:
    a. Sigmoid: The sigmoid function maps values to the range of [0,1] where hard negative numbers are mapped to 0 and hard positive numbers are mapped to 1. The sigmoid function is nice as it resembles allows use to look at it results as probability values, which is useful for many things, as well as gated operation in RNN networks. The sigmoid is not very popular activation function because it is easily saturated and causes vanishing gradients. This happens because many negative values are mapped to zero and therefore the gradient is zero, that is the gradients have vanished. In addition, sigmoid values are not zero centered. This causes the gradients in following layers to either be positive or negative and can greatly impact the learning.

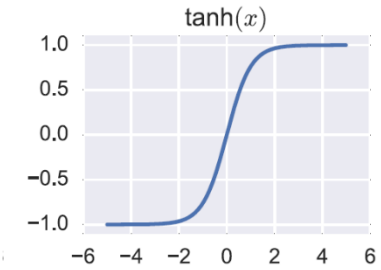
$$\sigma(x) = 1/(1+e^{-x})$$



b. Tanh: this activation function maps the values to the range of [-1,1]. It acts very similarly to sigmoid, in fact, it is just a scaled and translation of the sigmoid function ($\tanh = 2\sigma(2x) - 1$). The $tanh$ function suffers from the same problems as the sigmoid only that the $tanh$ is zero centered, and therefore is always better to use.

tanh

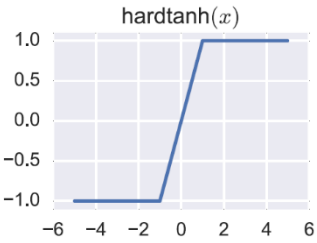
$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$



c. Hard Tanh: this is a variation on the $tanh$ function where a non-continuous version is used. Hard tanh is computationally cheaper than regular tanh but not differentiable at all points.

hard-tanh

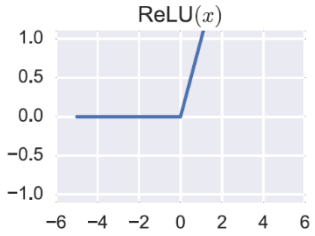
$$\text{hardtanh}(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise} \end{cases}$$



d. ReLU: (rectified linear unit) this function is very popular to use. It has some pros and cons. First ReLU has a non-saturable structure. This leads to acceleration of the SGD algorithm. In addition, it is very cheap computationally. On the other hand, this structure causes that the gradient to block the neuron when it gets into the zero part. This however, can be sometimes solved using the Leaky ReLU function.

ReLU (rectifier, rectified linear unit)

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases}$$



12. **Overfitting**: Overfitting is a situation where the model is too fit for the training data, that is it performs very well on the training data, but it performs poorly on the validation data. That is the model learned well however it failed to generalize. This can be dealt with using:
    a. Early Stopping: stop training when the performances on the validation data is dropping. Usually we would stop training based on the loss but only because the accuracy is hard to optimize on.
    b. More vs. Less Layers + Dimensions: Adding more layers/dimensions captures more complex structures but can easily overfit, less layers/dimensions captures more meaningful features however can be too little information to generalize

c. Regularization: In this technique a regularization term is added to the loss function. This discourages the network from learning a complex or flexible model and reduces the risk of overfitting. The added values depend on the parameters where R is usually taken as sum of absolute values. Since we are trying to minimize the loss, the process of optimization will try to fit small values to the parameters so the value of the regularization will not get too big. Regularization forces the parameter not to get too big.

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}} \left(\overbrace{\frac{1}{n}\sum_{i=1}^{n} L(f(\mathbf{x}_i;\Theta),\mathbf{y}_i)}^{loss} + \overbrace{\lambda R(\Theta)}^{regularization} \right)$$

$$R_{L_2}(\mathbf{W}) = ||\mathbf{W}||_2^2 = \sum_{i,j}(\mathbf{W}_{[i,j]})^2 \quad \text{L2 regularization}$$

$$R_{L_1}(\mathbf{W}) = ||\mathbf{W}||_1 = \sum_{i,j}|\mathbf{W}_{[i,j]}| \quad \text{L1 regularization}$$

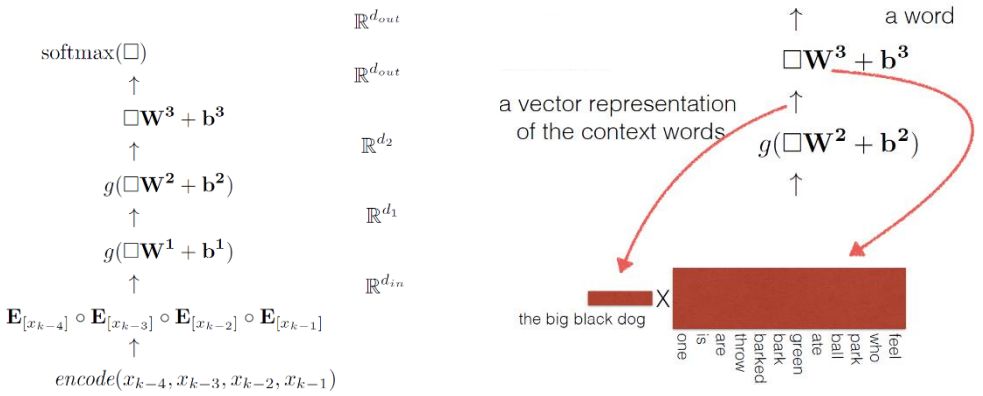
$$R_{\text{elastic-net}}(\mathbf{W}) = \lambda_1 R_{L_1}(\mathbf{W}) + \lambda_2 R_{L_2}(\mathbf{W}) \quad \text{elastic net}$$

13. Dropout: drop random neurons on each epoch. This help the network to generalize better because the weights cannot get to fit for specific task. Equivalent to training 2^n different networks.

$$\begin{aligned}\text{NN}_{\text{MLP2}}(\mathbf{x}) &= \mathbf{y}\\ \mathbf{h^1} &= g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})\\ \mathbf{h^2} &= g^2(\mathbf{h^1}\mathbf{W^2} + \mathbf{b^2})\\ \mathbf{y} &= \mathbf{h^2}\mathbf{W^3}\end{aligned} \quad \Rightarrow \quad \begin{aligned}\text{NN}_{\text{MLP2}}(\mathbf{x}) &= \mathbf{y}\\ \mathbf{h^1} &= g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})\\ \mathbf{m^1} &\sim \text{Bernouli}(r^1)\\ \tilde{\mathbf{h}}^1 &= \mathbf{m^1} \odot \mathbf{h^1}\\ \mathbf{h^2} &= g^2(\tilde{\mathbf{h}}^1\mathbf{W^2} + \mathbf{b^2})\\ \mathbf{m^2} &\sim \text{Bernouli}(r^2)\\ \tilde{\mathbf{h}}^2 &= \mathbf{m^2} \odot \mathbf{h^2}\\ \mathbf{y} &= \tilde{\mathbf{h}}^2\mathbf{W^3}\end{aligned}$$

14. Language Model: is the understanding of the word based on the context. It is often defined as $p(x_i|x_{1:i-1})$. To simplify this problem we use the Markovian Assumption dictating that the current prediction does not depends on information from the far past. That alows us to simplify the language model to be $p(x_i|x_{1-k:i-1})$. This is an ngram language model. By multiplying language models of different words we can have a model for a sequence.
15. Word encoding: There are different way to transform tokens into vectors so we can use them for calculations.
    a. One-Hot-Vector: in this approach, each vocabulary word is assigned an index. The vector representation of the word is a long vector where there is 1 in the index of the word. If we want to model a sequence, then there would be ones in multiple indices. For repeated words we can have the number greater than one, indicating the frequency of the word. This approach is very space consuming when (and it is usually the case) the vocabulary is very large. In addition, this approach doesn't have a notion of order. This can be solved by concatenating the one hot vectors.
    b. Embedding Layer: Associate each word with a word in a matrix E, where the row length is much smaller than the vocabulary size. This was we still get a representation of the word and in addition we don't have a problem with the size of the vocab since the output of E is always constant and smaller than the vocab size. E is initialized randomly or to some pre-trained embedding matrix and then fine-tuned as part of some bigger task. The fine-tuning makes the embedded representation of the words to be more specific for the task. Sequences are represented in the same way where the different word embeddings are either summed or concatenated.

16. Neural LM: this apprach uses multiple MLP layers on a concatenation of embedded words taken from a window in a sentence. The network is trained to predict the next word after the window, therefore in the last layer of the network, the weight matrix holds incoded representations of words. This ideas is useful because those word representations encode similarity traits in them and can help to better understand the n otion of words by machine. This approach is computationally expensive because we need to compute and normalize all vocabulary words for the softmax operation at each training step as well as due to large matrices (for example W^3 is of the size of the vocabulary). Note that in Neural LM row of matrix E represent words and in W^3 the columns represent words. In addition note that the vector fed into the last layer is a representation of the context. Can use to encode sentences.



17. Collobert and Weston: The idea is to use a center word, two previous words and two following words. Then replace the softmax function with a scoring function that is applied only on the five words used (the softmax normalizes over the whole data, expensive). Training is don by replacing the center word with a random word and computing its score but comparison to the score of the original word and use this as loss for training.

$$\text{score}(D, A, \square, C, F; G) = g(\mathbf{xU})\mathbf{v}$$
$$\mathbf{x} = (\mathbf{E}_{[D]} \circ \mathbf{E}_{[A]} \circ \mathbf{E}_{[G]} \circ \mathbf{E}_{[C]} \circ \mathbf{E}_{[F]})$$
$$L(w, c, w') = max(0, 1 - (\text{score}(c_{1:k}; w) - \text{score}(c_{1:k}; w')))$$

18. Word2Vec: is a two-layered neural network that offers improvements on Collobert and Weston by using some tricks to speed the computation and to give better representations to the different tokens. The model learns the word embeddings by training on predicting the next token, at the end of the training we have the weights initialized to represent the words. We remove the prediction layer and can now output an embedded representation of each word by passing it through the network. Word2Vec exploits syntactic differences when used with small window, and sematic differences when used on a larger window. The word2vec applies the following ideas:
    a. Use a different scoring function than Collobert and Weston based on probabilities. We are trying to figure out what is the probability that the five-token sequence came from the corpus. s is a learnable score function.

$$P(D = 1|w, c) = \sigma(s(w, c)) = \frac{1}{1 + e^{-s(w,c)}}$$

b. Negative Sampling: this approach boosts the accuracy of the trained vector representation. The idea is to base the loss function on the true label but also not false ones. The loss is now the sum of the sequences I have labeled correctly as being part of the corpus and the sequences I have labeled correctly as not part of the corpus.

good word+context pairs **bad** word+context pairs

$$P(D=1|w,c)=\sigma(s(w,c))=\frac{1}{1+e^{-s(w,c)}}$$

$$\mathcal{L}(\Theta;D,\bar{D})=\sum_{(w,c)\in D}\log P(D=1|w,c)+\sum_{(w',c)\in\bar{D}}\log P(D=0|w',c)$$

$$P(D=0|w,c)=1-P(D=1|w,c)$$

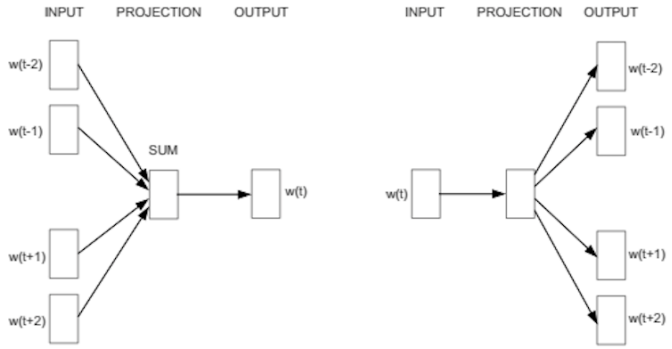
$$P(D=1|w,c)=\sigma(s(w,c))=\frac{1}{1+e^{-s(w,c)}}$$

$$\mathcal{L}(\Theta;D,\bar{D})=\sum_{(w,c)\in D}\log P(D=1|w,c)+\sum_{(w',c)\in\bar{D}}\log P(D=0|w',c)$$

$$\mathcal{L}(\Theta;D,\bar{D})=\sum_{(w,c)\in D}\log P(D=1|w,c)+\sum_{(w',c)\in\bar{D}}\log P(D=0|w',c)$$

c. Hierarchical Softmax: a technique used to calculate the softmax faster by defining the probability of a word to be the probability of a random walk from the root to a leaf in a binary tree where the words are leaves. $O(|V|)\rightarrow O(\log(|V|))$.

d. CBOW / Skip-Gram: word2vec uses those two approaches interchangebly. While in CBOW the model tired to predict the center word based on the context, the idea behind skip gram is to predict the k words around a centered word instead. The drawback with CBOW is that it treats the entire context as one observation. We assume that the probabilities are independent of each other, enabling us to treat the probability as a multiplication of the independent probabilities. This allows to simplify the score function.



CBOW

$$\text{score}(w;c_1,\ldots,c_k)=(\sum_{i=1}^{k}\mathbf{E}_{[c_i]})\cdot\mathbf{E}'_{[w]}$$

$$P(D=1|w,c_{1:k})=\frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{c_1}+\mathbf{w}\cdot\mathbf{c_2}+\ldots+\mathbf{w}\cdot\mathbf{c_k})}}$$

Skip-grams:

$$P(D=1|w,c_i)=\frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{c_i}}}$$

$$P(D=1|w,c_{1:k})=\prod_{i=1}^{k}P(D=1|w,c_i)=\prod_{1=i}^{k}\frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{c_i}}}$$

$$\log P(D=1|w,c_{1:k})=\log\sum_{i=1}^{k}\frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{c_i}}}$$

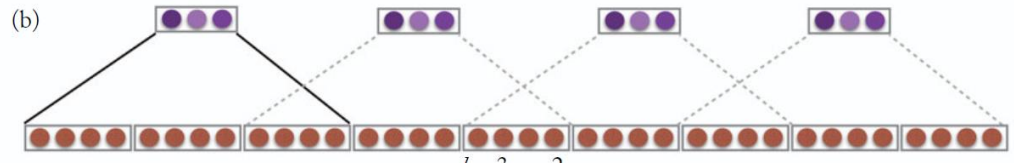
$$\mathcal{L}(\Theta;D,\bar{D})=\sum_{(w,c)\in D}\log P(D=1|w,c)+\sum_{(w',c)\in\bar{D}}\log P(D=0|w',c)$$

$$=\arg\max_{\theta}\sum_{(w,c)\in D}\log\frac{1}{1+e^{-v_c\cdot v_w}}+\sum_{(w,c)\in D'}\log(\frac{1}{1+e^{v_c\cdot v_w}})$$

19. **Fast Text**: This approach uses sum of character ngram embeddings as the word representation.
20. **Calculating Similarity Between Words**: using word embeddings:
    a. Similarity between two vectors: cosine similarity.
    b. Top k similar: apply vector multiplication to get the doc product value, then rank. This should be very fast.
    c. Find similarity to a group of words: sum the word vectors and calculate similarity according to that.
21. **Gradient Based Training**: The training process relies on the computation of gradients to find an optimum of the function. Thanks to the chain rule, we can construct simple building block, which we can calculate the gradient function before hand and then just use it as plug and play like LEGO blocks for training complicated neural networks.
22. **Feature Embeddings**: A method where features are represented as using embedding vectors. We would like that similar features receive similar representations.
23. **Concatenation vs. Summation Representation**:
    a. Concatenation: Concatenation of features is very effective. It allows for vectors of different representation to be used together, the "role" of each representation does not change (the next layer can easily learn to retrieve it) and it keeps the notion of order between the features. However, only a fixed number of such representation can be used since the next layer expects an input vector of fixed size.
    b. Sum: This allows us to treat as many vectors as we need, however they all must be of the same dimension. In this approach it is very difficult to reconstruct the different features inserted to it.
24. **CBOW**: (Continuous Bag of Words) is summed based approach. Using this approach, we can give each feature a different weight.

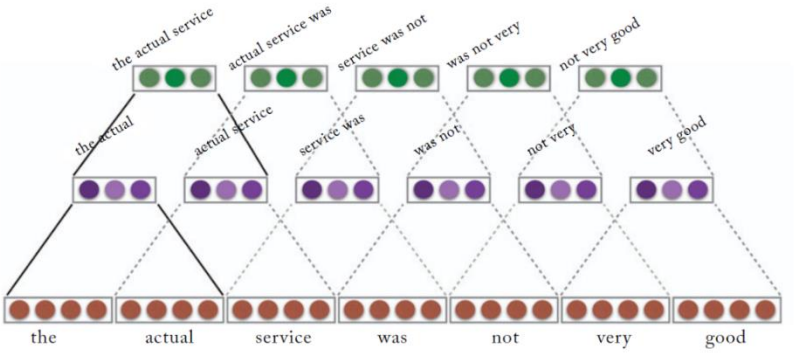
$$CBOW(f_1,...,f_k) = \frac{1}{k}\sum_{i=1}^{k} v(f_i) \qquad WCBOW(f_1,...,f_k) = \frac{1}{\sum_{i=1}^{k} a_i}\sum_{i=1}^{k} a_i v(f_i)$$

25. **Hybrid Representation**: The idea is to use the pro's in both methods. In hybrid representations we concatenate and sum the features in a window separately, then the two are concatenated.
26. **ConvNet**: Convolution networks allow to identify informative local predictors. That is, given a sequence of features, ConvNets are useful for creating another feature vector that represents the 'big idea' in the sequence of features. This helps to reduce the dimension of the feature space as well as extract the important features. The convolution is achieved by convolving different kernels with the features, where those kernels are specifically engineered to extract some specific features. In Computer Vision, ConvNets are used to extracts object positions in images. Research show that the ConvNets learn to recognize edges of objects, then shapes, and so on until they extract the object. In NLP, a sentence is convolved with multiple kernels to produce summarized vectors which are later summed/pooled/joined in some other way to generate a single encoded vector that encodes the important features in the data. In addition, convolution is a simple matrix multiplication and therefore it is very computationally efficient when using a GPU.
    Strides: this is how many step are skipped on each step of the convolution, where 'step' can



be a pixel, a character, a token, etc.

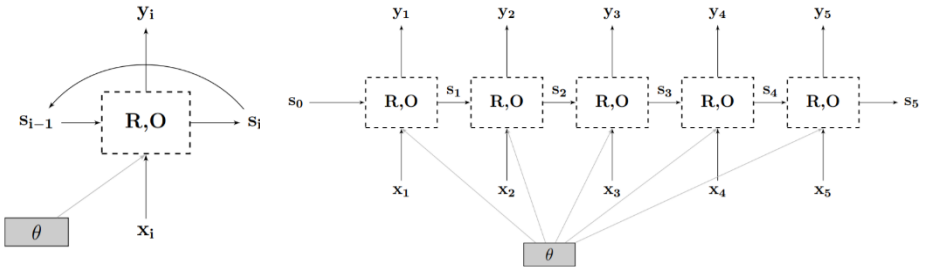
Hierarchy: we can apply convolution on top of a convoluted layer. This gives us a bigger covrage of the total sequence.



Dilated Convolution: when strides and hierarchy are used on a sequence. Using this approach we can have the different layers extract features form features created from tokens it hadn't seen. T

27. Pooling Layer: A matrix is created by concatenating the vectors verically. Then the i^{th} value in the pooled vector is the output of some function applied on the i^{th} row. Some examples for this function can be the sum or max functions. Max-pooling layer acts as a reginal 'main feature' extractr. One advantage of specifically using max pooling is that it is easy to trace back to the origin of each value.
28. Hashing Trick: When we have large vocabulary but limited memory, we can use hashing to store the large vocabulary. When collision occurs (since the memory is too small), the value is replaced. Since the vocab is so big, this overriding should not acutally matter. We can improve this solution by hashing to a bag of words.

29. Non-RNN approaches for dealing with sequences:
    a. For fixed sequence size, the tokens can be concatenated to get a single vector
    b. CBOW
    c. Break a sequence into fixed sized windows and then use (a)
    d. Use ConvNet nets to generate a single vector
30. RNN: is a network that uses memory that is passed between the cells to be able to remember information from previous tokens in the sequence and thereby achieves global order of elements. The network is defined recursively, where the parameters are share across time.



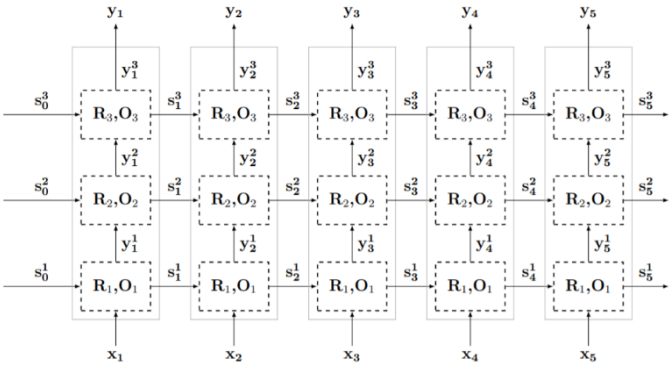
$$RNN(\mathbf{s_0}, \mathbf{x_{1:n}}) = \mathbf{s_n}, \mathbf{y_n}$$
$$\mathbf{s_i} = R(\mathbf{s_{i-1}}, \mathbf{x_i})$$
$$\mathbf{y_i} = O(\mathbf{s_i})$$
$$\mathbf{x_i} \in \mathbb{R}^{d_{in}},\ \mathbf{y_i} \in \mathbb{R}^{d_{out}},\ \mathbf{s_i} \in \mathbb{R}^{f(d_{out})}$$

Classification: RNNs can be used for classification tasks, for example we can predict the next word in a sequence based on all previous tokens:

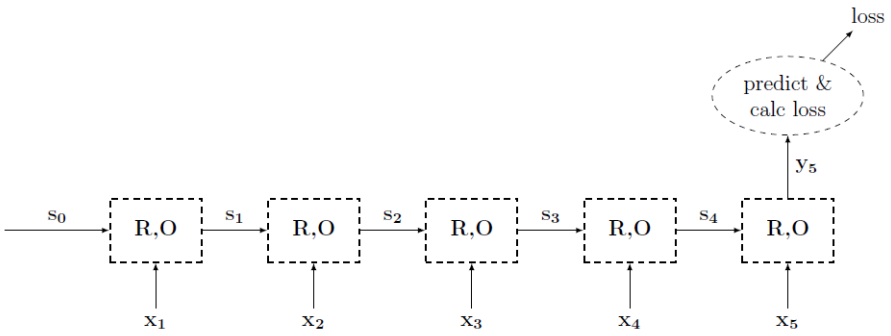
$$predict(\mathbf{x_{1:n}}) = \arg\max_i P(\mathbf{x_{1:n}}|RNN_i)$$

$$P(\mathbf{x_{1:n}}|RNN) = \prod_{j=1}^{n} softmax(MLP(RNN(\mathbf{x_{1:j}})))_{[x_j]}$$

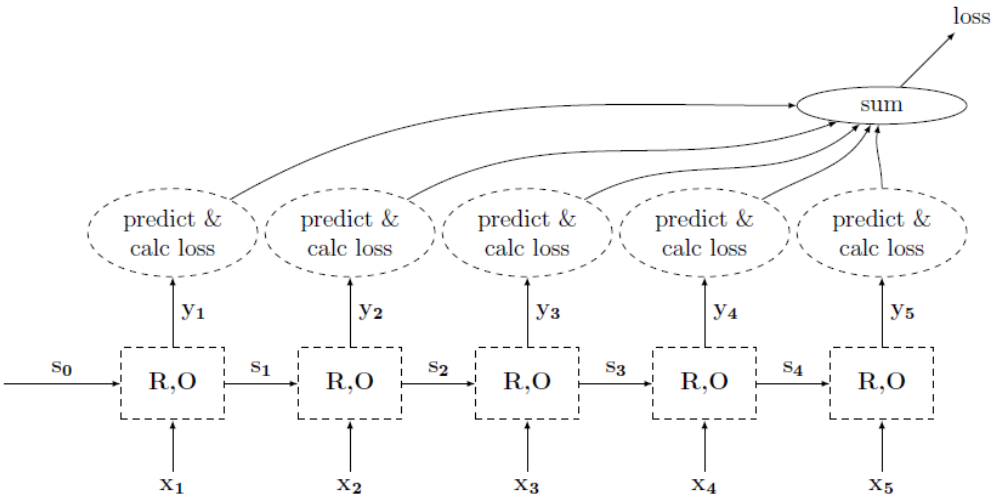
Deep RNNs: RNN can also be stack on top of each other to create deep RNNs.



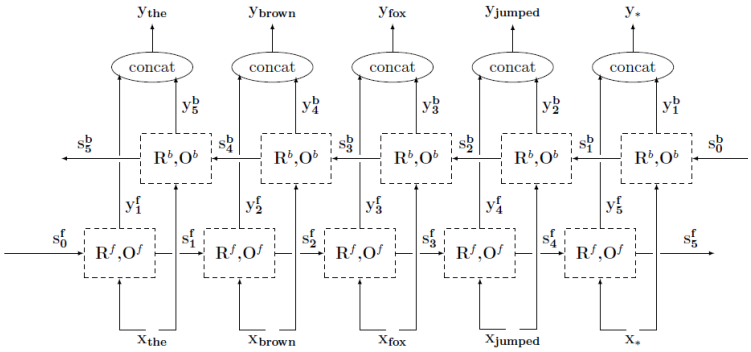
31. Acceptor: (n to 1) only the last output of the RNN is used, after the whole sequence is fed. The prediction task is based on this single output. The network learn by backpropagating through the whole network.

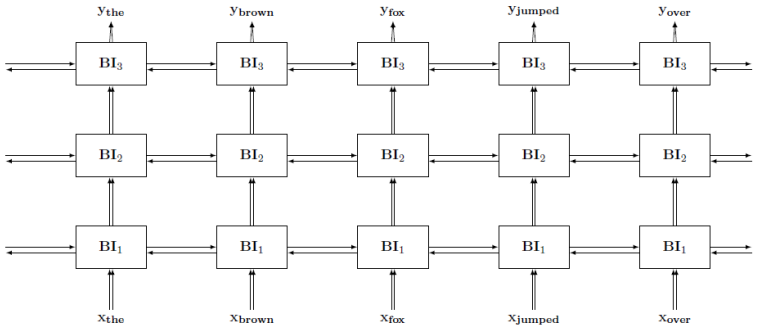


32. **Transducer**: (n to n) all the outputs of all the RNN cells are used to perform the prediction task. A sum of the loss on each of those outputs is used to train the network with backpropagation.



Variations of architecture include a bi-directional approach. This helps to take future information into account. In this case the encoded vector of the current token in the sequence is a concatenation of the output vectors of the two directions. In addition multiple layers can be stacked to create deep RNNs or deep Bi-RNNs (See LSTM).

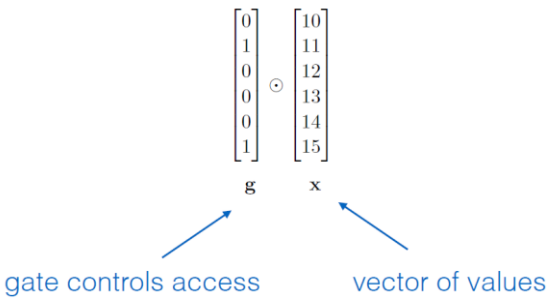


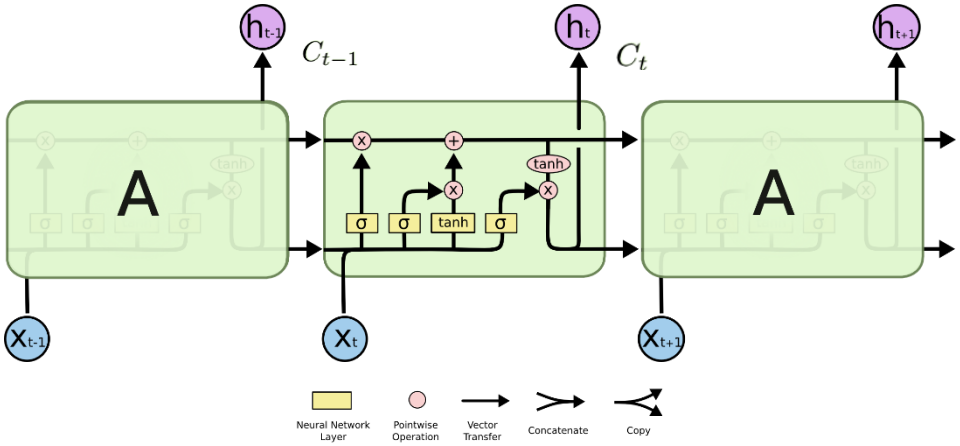


33. Simple RNN (Elman's RNN): A very simple implementation of an RNN with non-linearities. This is theoretically enough for an effective RNN however due to vanishing gradients, it performs poorly. In addition the whole memory is written on every step of the network.

$$R_{SRNN}(\mathbf{s_{i-1}}, \mathbf{x_i}) = tanh(\mathbf{W^s} \cdot \mathbf{s_{i-1}} + \mathbf{W^x} \cdot \mathbf{x_i})$$

34. LSTM: is a type of RNN which yeilds the best RNN based results. The LSTM is designed to be able to learn long term dependencies by controling the 'memory access' using dynamic gates. We can treat each index in a vector as a memory cell. Controlling those cells is dones using weights in the rage of 0 and 1 and act a gates to how much information should be passed into the cell LSTMs can learn some heirachial patterns, but not all. In addtion, LSTMs solve the problem of vanishing gradients. In addition, note that in LSTM all the matrix multiplications can be performed in one multiplication operation.





$$
\begin{aligned}
R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) &= [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} &= \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} &= \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} &= \sigma(\mathbf{W^{xi}} \cdot \mathbf{x_j} + \mathbf{W^{hi}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{f} &= \sigma(\mathbf{W^{xf}} \cdot \mathbf{x_j} + \mathbf{W^{hf}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{o} &= \sigma(\mathbf{W^{xo}} \cdot \mathbf{x_j} + \mathbf{W^{ho}} \cdot \mathbf{h_{j-1}}) \\
\mathbf{g} &= \tanh(\mathbf{W^{xg}} \cdot \mathbf{x_j} + \mathbf{W^{hg}} \cdot \mathbf{h_{j-1}}) \\
O_{LSTM}(\mathbf{s_j}) &= O_{LSTM}([\mathbf{c_j}; \mathbf{h_j}]) = \mathbf{h_j}
\end{aligned}
$$

The LSTM is designed as follows:

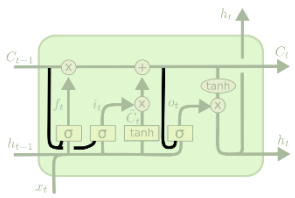
a. Cell state belt: $(C_{t-1} \rightarrow C_t)$ This line is responible for passing the state bettwen the cells, that is, it is designed to remember patterns from the previous tokens. This line has minimal interaction with other component. Addition and subtraction of information is allowed, however only after is was carefully regulated by gates which decide how much information can get through.

b. Forget Gate Layer: $(x_t, h_{t-1} \rightarrow f_t)$This gate is resposible for deciding what information show be removed from the cell state based on h_{t-1} and x_t. The cell generates a vector of values between 0 and 1, which act as weights when multiplied by the state vector. Note that h_{t-1} is the output of the previous cell and x_t is the t^{th} input in the sequence.

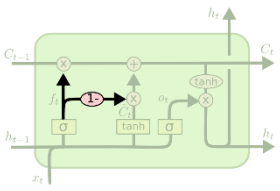
c. Remember Layer: $(x_t, h_{t-1} \rightarrow i_t, \widetilde{C_t})$ This layer is responible for deciding what new information should be stored in state vector. x_t and h_{t-1} are used to decide which values should be updated and and to create a new candidate vector. The first part (i_t) is achieved by passing through a sigmoid gate. This decides which values should be updated by generating weight (like in (a)). The latter $(\widetilde{C_t})$ is achived by passing through a $tanh$ function. The two resulting vectors are multiplied, that is the weights from the first part are applied to the candidate vector. This represents the 'new information' that should be saved, and therefore it is added to the cell state.

d. Output Generation: The output is generated based on the new cell state, where x_t and h_{t-1} are used to determine what parts of it should be filtered. To achieve this we first use another sigmoid gate on x_t and h_{t-1} to generate the weights, and then multiply it by the cell state vector (which was passed through a $tanh$ function to keep the values between -1 and 1) to produce the final filtered output.

e. Veriations: there are multiple implementations of the LSTM which add on the basic one described in parts (a-d). One approach adds "peephole connections" which connect the cell state to some or all of the sigmoid function, allowing for the network to take the current state into account (left image). Another approach is to use the same filtring for both the Removing and the Remembering operations by considering the complementary weights (right image). This creates a condition where we forget only when we are about to input something in its place or vise versa.

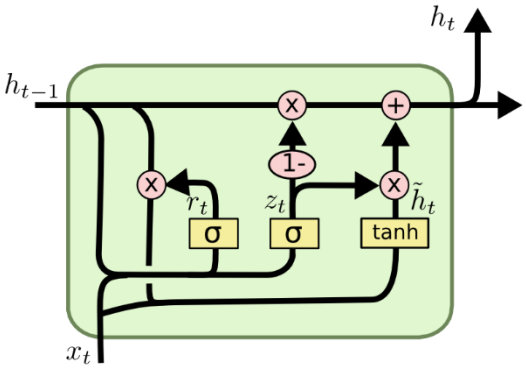


$$
\begin{aligned}
f_t &= \sigma\left(W_f \cdot [C_{t-1}, h_{t-1}, x_t] \; + \; b_f\right) \\
i_t &= \sigma\left(W_i \cdot [C_{t-1}, h_{t-1}, x_t] \; + \; b_i\right) \\
o_t &= \sigma\left(W_o \cdot [C_t, h_{t-1}, x_t] \; + \; b_o\right)
\end{aligned}
$$



$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$

35. **GRU**: An RNN that acts similarly to an LSTM however was developed without the knowlage about the existing LSTM. In GRU the 'forgert' and 'remember' gates are combined into a single update gate. In addition the cell state and the output (hidden) vector are the same. This resulted then a simpler model than the LSTM.

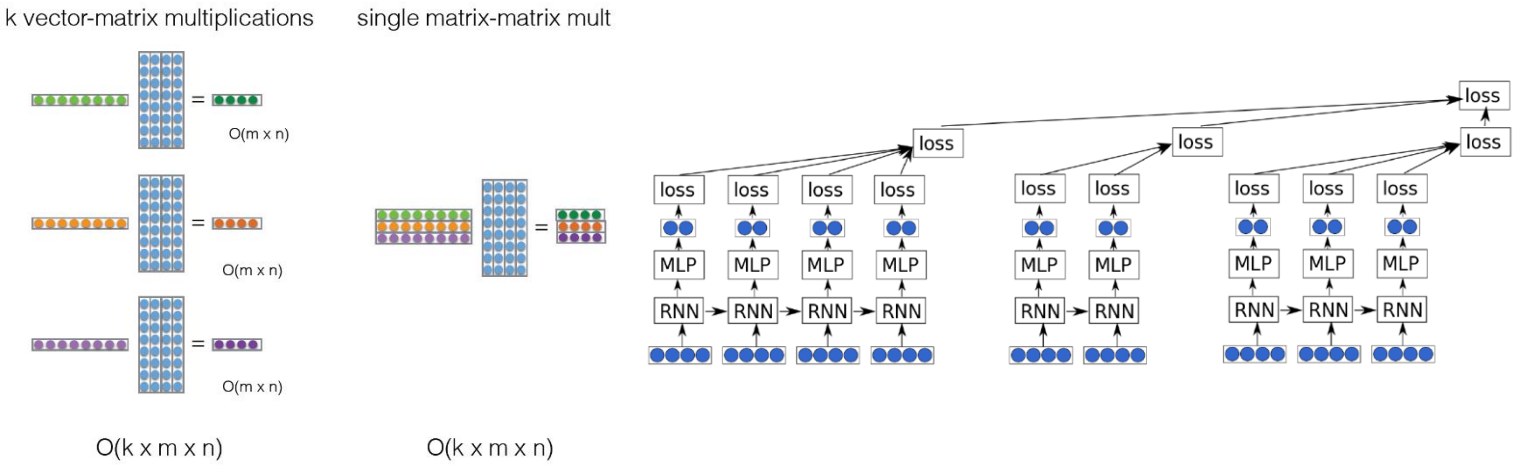


$$s_j = R_{GRU}(s_{j-1}, x_j) = (1 - z) \odot s_{j-1} + z \odot \tilde{s}_j$$
$$z = \sigma(x_j W^{xz} + s_{j-1} W^{sz})$$
$$r = \sigma(x_j W^{xr} + s_{j-1} W^{sr})$$
$$\tilde{s}_j = \tanh(x_j W^{xs} + (r \odot s_{j-1}) W^{sg})$$

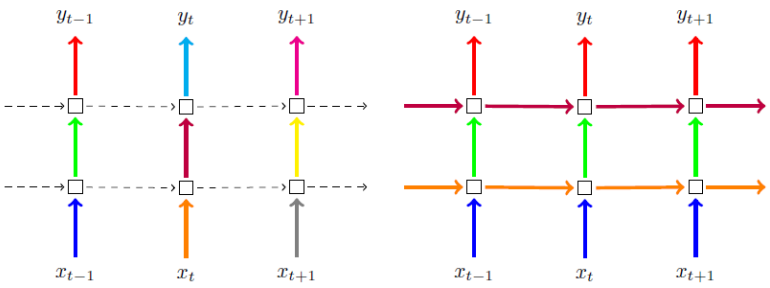
GRU vs. LSTM:

a. In GRU the previous state is considered.
b. In GRU both the output and the state are the same.
c. In GRU the forget and remember are both dependent on each other since they use the same gate.
d. In GRU the candidate vector is gated by the previous state.
e. LSTMs can count and GRUs cannot. (?)

36. **RNN Batching**: batching can greatly improve computation time, especially when running on GPU. Since all we are doing is multiply vectors by matrices and passing the results into activation functions, we can simply extend the input vector notation to input matrix notation where each row j in the i^{th} matrix represents the i^{th} token in the j^{th} sequence.
When applying batching to RNN we may stumble upon the problem of uneven sequences. This problem can be solved either by padding the shorter sequences and shorten other to match some fixed size. Another approach is to group the sequences by their length and build a separate LSTM for each such batch size. Then pass batches of some size in their corresponding LSTM graphs.

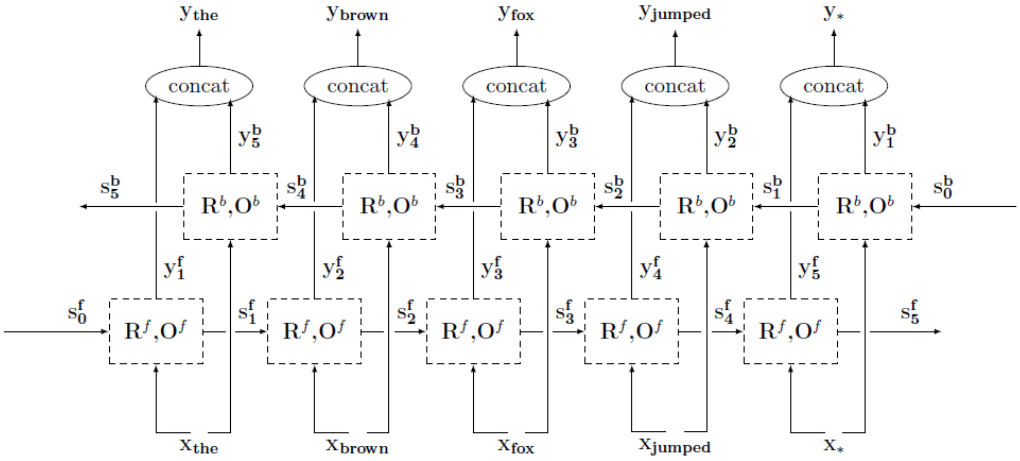


37. **RNN Dropout**: we can apply dropout on RNNs as well. There are multiple approaches on how to perform this. The main issue is centered around the requirement that the state will always

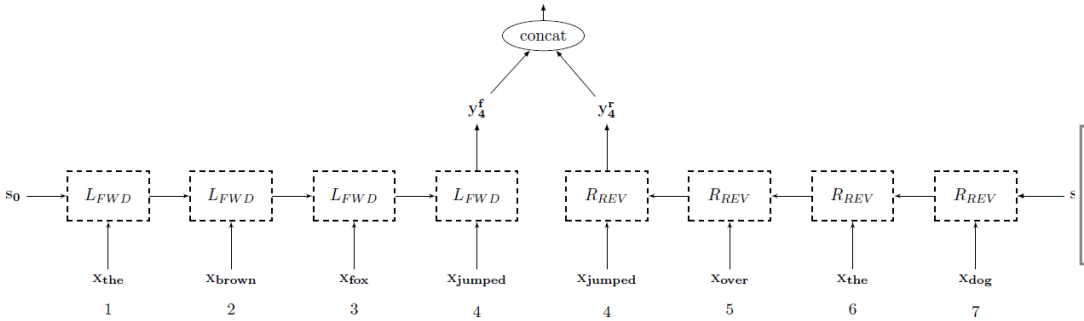


pass to the next cell, otherwise there is no use in this architecture. Below are two possible approaches: In the first (right) each cell output is dropped randomly but the state vector is never droped. Another approach (right) does allow to drop state vectors, but this requires to drop the whole sequence of state vectors for the layer, and in addition single outputs cannot be dropped.

38. **Bi-LSTM**: bi-directional LSTM is an RNN which improve the LSTM architecture for tasks where the model can benefit from future knowlage. In this architecure there are two LSTMs in play, where one runs from the beginning of the seque to the end (normally) and the other run from the end of the sequence to the beginning. This provides an infinite (non-markovian) window around a word, which is very useful for sequence tagging problems.

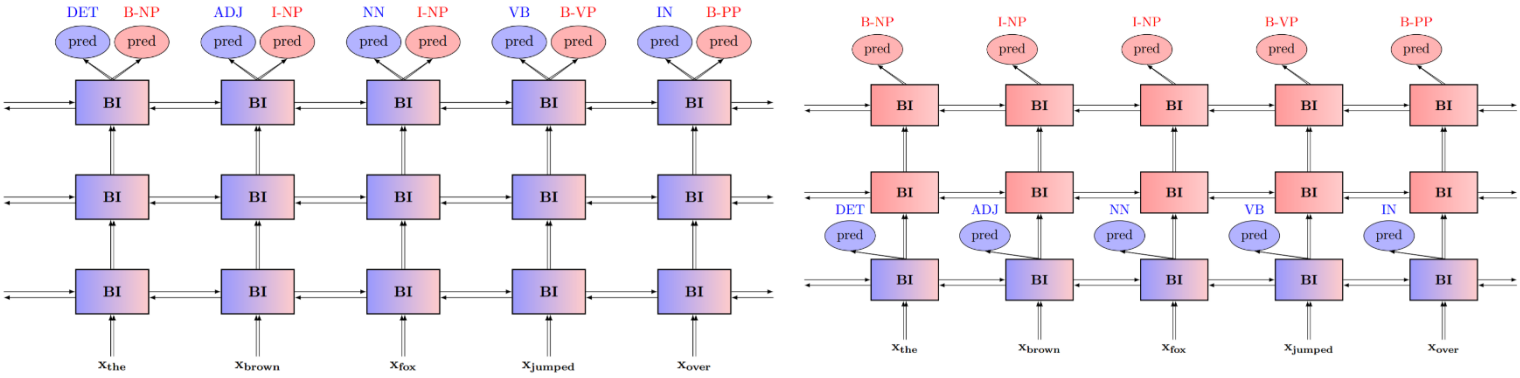


In most cases the results of the different direction are concatenated to create the vector representation of the word at that position. This gives us a representation of the word based on the previous and the future tokens. The downside of LSTMs is that they need great amounts of data to train on.

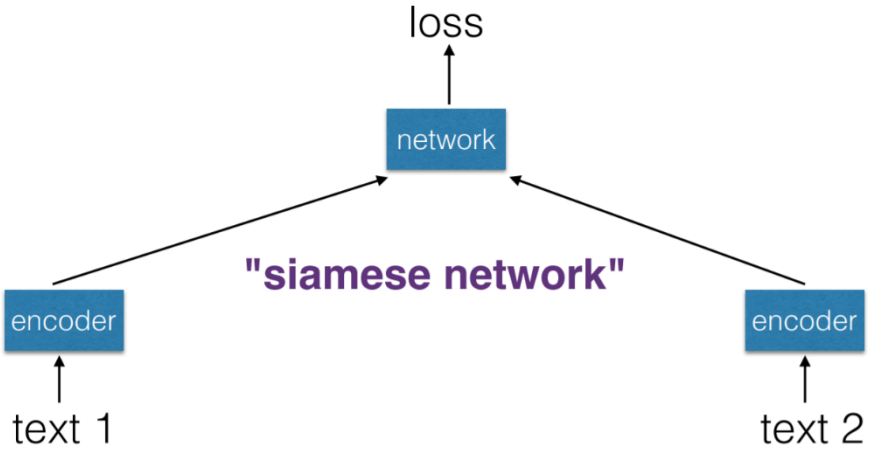


$$BiRNN(\mathbf{x_{1:7}}, 4) = [\mathbf{y_4^F}; \mathbf{y_4^R}]$$
$$\mathbf{y_4^F} = RNN_F(\mathbf{x_{1:4}})$$
$$\mathbf{y_4^R} = RNN_R(\mathbf{x_{7:4}})$$

39. Multi-Task Learning: Many tasks have shared tructures, and therefore predition on one problem can help predicting another problem. The approach is to train the network on multiple tasks. This problem is not so easy to apply, and often it does not yield better results. This problem is greatly limited by the dimention of the hidden layers, where the vectors need to hold information about both tasks.
    Another approach is to apply multi-task learning on tasks that are heirachially dependent, that is the output of one problem can help determine the output of the other task. In this case the lower layers are trained on one task, and the upper layers are trained on another but also fine-tuned on the whole problem.



40. Siamese Network: in this type of problems we pass two different input sequences through the same encoder, and then predict based on the two. The name derives from the symmetry in the architecutre.



41. **Generators**: networks that can generate something.
42. **Conditioned Generators**: generators where the output is condition on complex input.
43. **Sequence Generator:** (0 to n) generator that generates a sequence
44. **RNN-generator**: a generator that generates a sequence by trying to predict the next token in the sequence. That is, in time i choose the most probable token t_i based on the distribution of the input and the previous tokens, that is based on $p(t_i = k|t_{1:i-1})$.
    Note that this process utilize 'start' and 'end' symbol, where the 'start' symbol is fed into the first cell and the 'end' symbol should be outputted by the last cell signaling that the computation should stop.

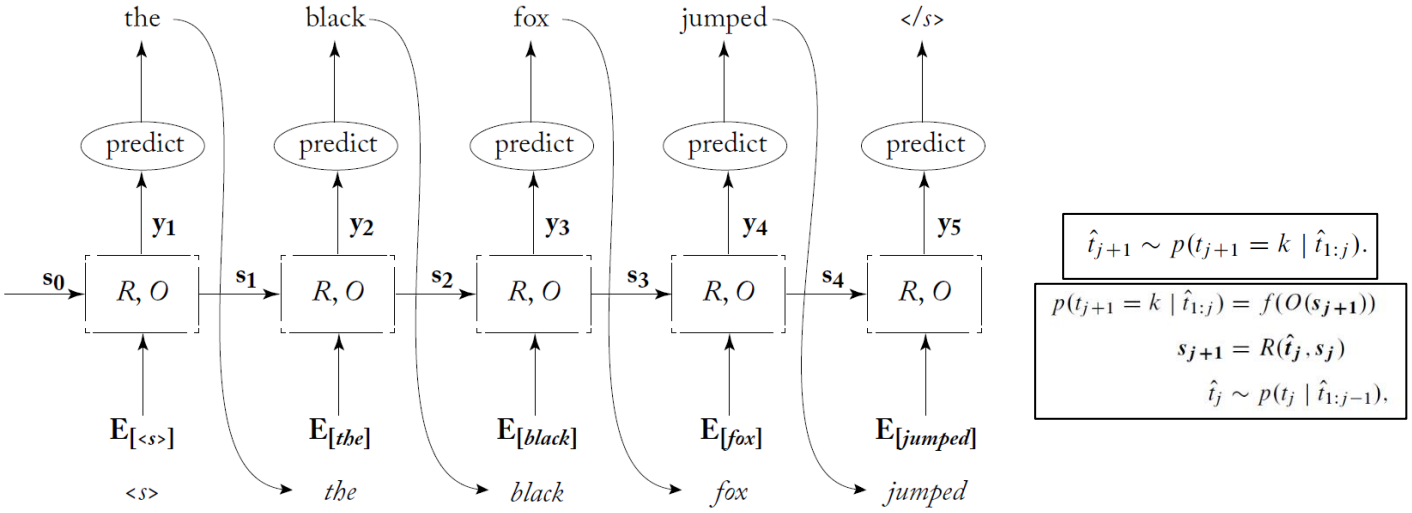
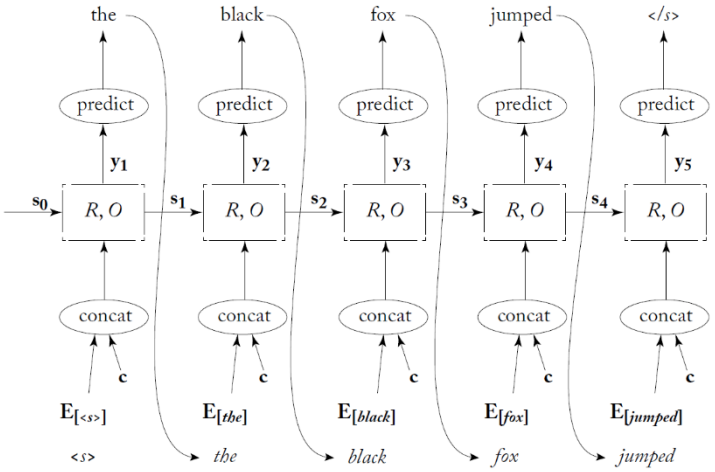


Figure 17.1: Transducer RNN used as a generator.

45. **Teacher-forcing**: the method used to train RNN-generators, where the RNN trains on the tokens in the sequence, where it starts with an input of a new sentence and is supposed to predict the next token, and the loss is computed accordingly. A disadvantage of this approach is that it does not handle cases where the token was unseen before and therefore has no weights tuned for it leading for poor performance in such scenarios.
    Mathematically: Yoav's book: p.197

46. **RNN-Conditioned-Generation**: Here the concept of 'conditioning-context' I introduced, where a vector c is also fed into the cell, however it is fed to all time states. The c vector should hold hits that could help the network get better. For example, c could indicate the topic of the text. Mathematically: Yoav's book: p.197.
Disadvantage – c is of fixed length and therefore has hard time holding all the information needed for the decoding process.



$$\hat{t}_{j+1} \sim p(t_{j+1} = k \mid \hat{t}_{1:j}, c).$$

$$p(t_{j+1} = k \mid \hat{t}_{1:j}, c) = f(O(\boldsymbol{s_{j+1}}))$$
$$\boldsymbol{s_{j+1}} = R(\boldsymbol{s_j}, [\hat{\boldsymbol{t}}_{\boldsymbol{j}}; \boldsymbol{c}])$$
$$\hat{t}_j \sim p(t_i \mid \hat{t}_{1:j-1}, c).$$

Figure 17.2: Conditioned RNN generator.

47. **Conditioning-Context**: See **RNN-Conditioned-Generation**
48. **Sequence-to-Sequence Conditioned Generation (encoder-decoder)**: In this approach we take c as an encoded sequence. We are interested to generate and output sequence $t_{1:m}$ based on an input sequence $x_{1:n}$. In this approach the input sequence in encoded and then fed as conditioned input to a decoder network. Often an RNN is used for encoding as decoding. This is useful for generating sequences of length n to length m. Seq2Seq methods are also useful for other problem setups such as image and text in image captioning. Other uses are possible.

$$\boldsymbol{c} = \mathrm{RNN}^{\mathrm{enc}}(\boldsymbol{x_{1:n}}).$$

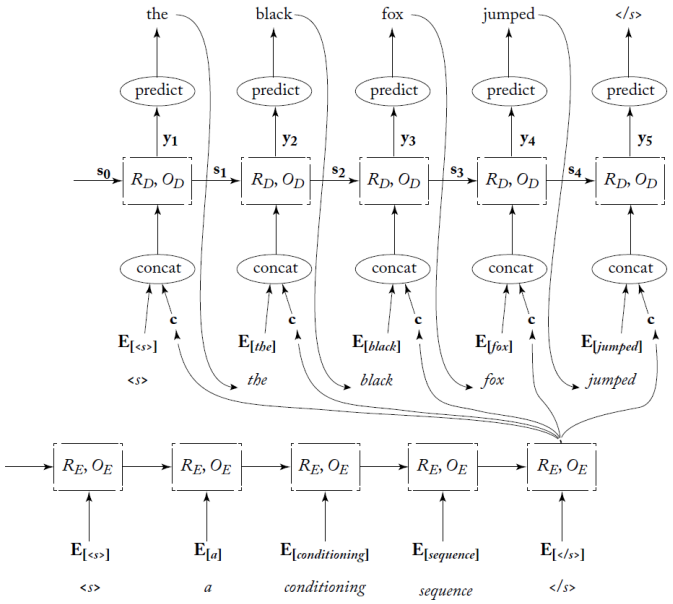


Figure 17.3: Sequence-to-sequence RNN generator.

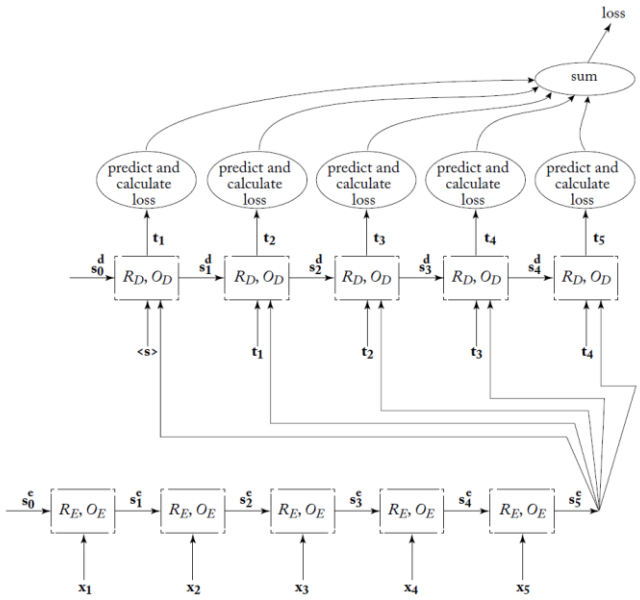
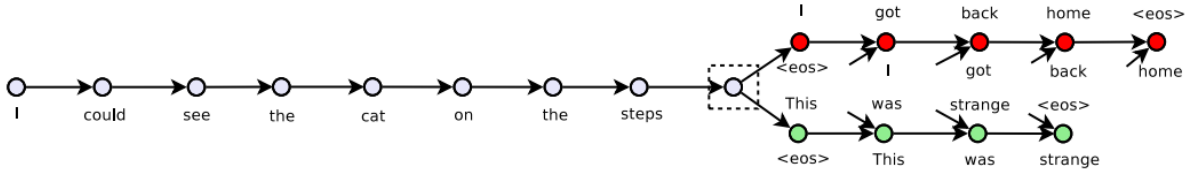


Figure 17.4: Sequence-to-sequence RNN training graph.

49. **Auto Encoder**: In the search for training good encoders, one can encode a sequence, then try to reconstruct the original sequence. Ex: sentence auto-encoding.
50. **Skip Thoughts (Paper)**: Sentence meaning is based on the context. Step 1: encode the sentence. Step 2: decode – one decoder is trained to decode the previous sentence and the other is trained to decode the next sentence. Step 3: propagate the error through the whole network. Finally the sentence representation is said to be the output of the encoder.



51. **Conditioned Generation with Attention**: The encoder outputs a **sequence** of vectors and <u>soft attention mechanism</u> that **learns** which vectors should get more weight, that is which parts of the output sequence should get more focus. The weighted vector is used as the conditioned vector c, fed into the decoder. The conditioned vector is computed independently from the encoding mechanism which still needs to run only once per input sequence. The weights may change for each decoding stage.
    The attention function is defined to produce a weighted averaged vector however we want to it to be a trainable parametrized function. We can therefore learn the coefficients deciding on the weight each vector receives. Each coefficient is defined as an MLP with softmax that receives the state parameter and the encoded vector i which the weight is supposed to be applied on. This way the weights can be learned for each stage of the decoder together with the whole network.
    We are not required to add an encoding layer before the attention, however such later helps in giving better representation of the input, and in addition is add more complexity to the network, allowing it to perform more complicated deduction mechanisms. For example the encoding mechanism may encode the positional information, and the attention later can give more wright to the elements in the beginning of the sequence (if they are more important). Attention mechanism offers good visualization since one can draw the weight given for evey encoded representation of every input and correlate it with the output token.

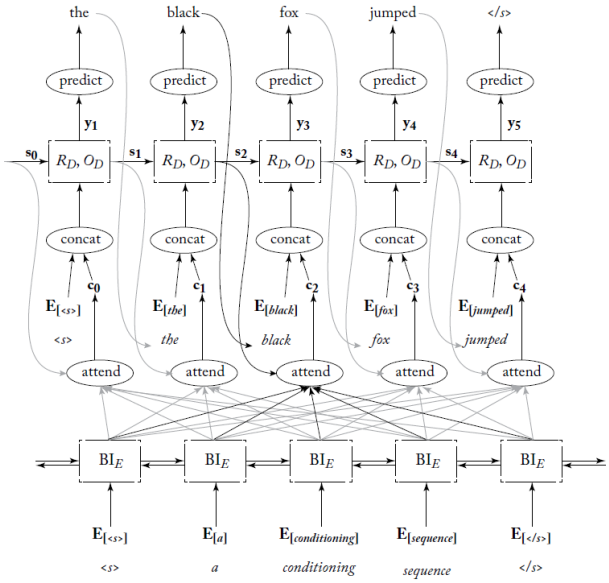


Figure 17.5: Sequence-to-sequence RNN generator with attention.

$$
\begin{aligned}
p(t_{j+1} = k \mid \hat{t}_{1:j}, \boldsymbol{x_{1:n}}) &= f(O_{\text{dec}}(\boldsymbol{s_{j+1}})) \\
\boldsymbol{s_{j+1}} &= R_{\text{dec}}(\boldsymbol{s_j}, [\hat{\boldsymbol{t}}_{\boldsymbol{j}}; \boldsymbol{c}^{\boldsymbol{j}}]) \\
\boldsymbol{c}^{\boldsymbol{j}} &= \sum_{i=1}^{n} \boldsymbol{\alpha}^{\boldsymbol{j}}_{\boldsymbol{[i]}} \cdot \boldsymbol{c_i} \\
\boldsymbol{c_{1:n}} &= \text{biRNN}^{\star}_{\text{enc}}(\boldsymbol{x_{1:n}}) \\
\boldsymbol{\alpha}^{\boldsymbol{j}} &= \text{softmax}(\bar{\boldsymbol{\alpha}}^{\boldsymbol{j}}_{\boldsymbol{[1]}}, \ldots, \bar{\boldsymbol{\alpha}}^{\boldsymbol{j}}_{\boldsymbol{[n]}}) \\
\bar{\boldsymbol{\alpha}}^{\boldsymbol{j}}_{\boldsymbol{[i]}} &= \text{MLP}^{\text{att}}([\boldsymbol{s_j}; \boldsymbol{c_i}]) \\
\hat{t}_j &\sim p(t_j \mid \hat{t}_{1:j-1}, \boldsymbol{x_{1:n}}) \\
f(\boldsymbol{z}) &= \text{softmax}(\text{MLP}^{\text{out}}(\boldsymbol{z})) \\
\\
\text{MLP}^{\text{att}}([\boldsymbol{s_j}; \boldsymbol{c_i}]) &= \boldsymbol{v} \tanh([\boldsymbol{s_j}; \boldsymbol{c_i}]\boldsymbol{U} + \boldsymbol{b}).
\end{aligned}
$$

Different types of attention functions:

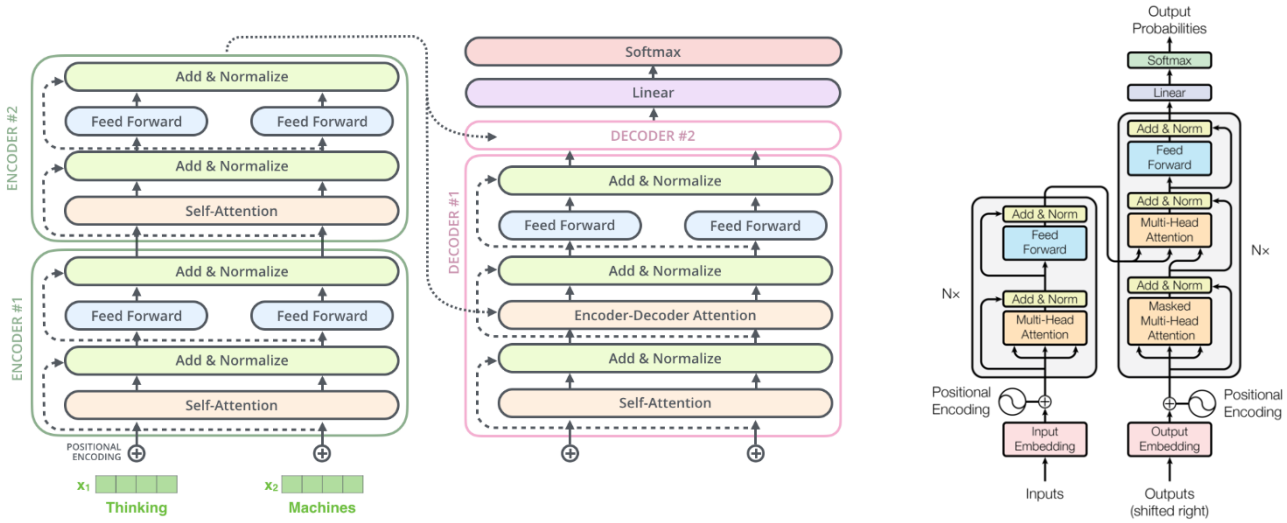
	PROs	CONs
MLP	Parameters are trainable	Slow and memory consuming
Dot product	Fast, still multiplies by weights	Simpler than trained weights
Scaled dot product	Helps for stabilization of the gradients	
Biaffine Transform	Improvement on the dot product	

Soft vs. Hard Attention:
In soft attention we are computing a weighted averaged vector from the encoded outputs. In hard attention one vector is sampled and the coefficient are considered as the sample rate. Learning of the coefficient is done using Mote-Carlo method, that is making 0-1 decision for each word: should include in or should we not include it as part of the weighted averaged vector.
Monotonic Attention:
Sometimes we have monotonic relation between the input and the output. This gives us more information.

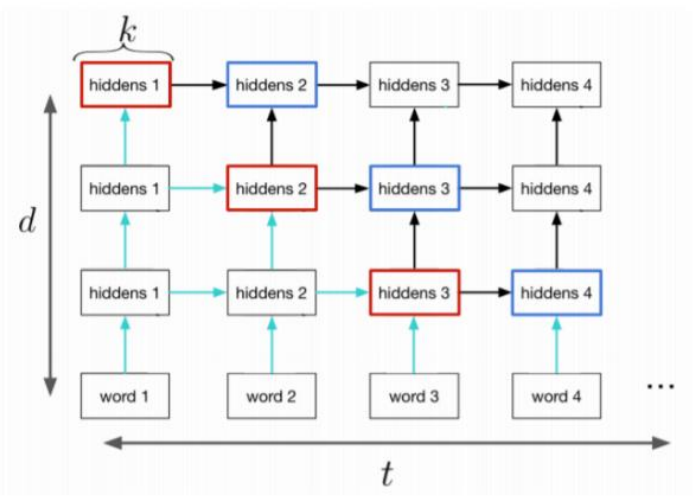
See Yoav's book, p.204-208

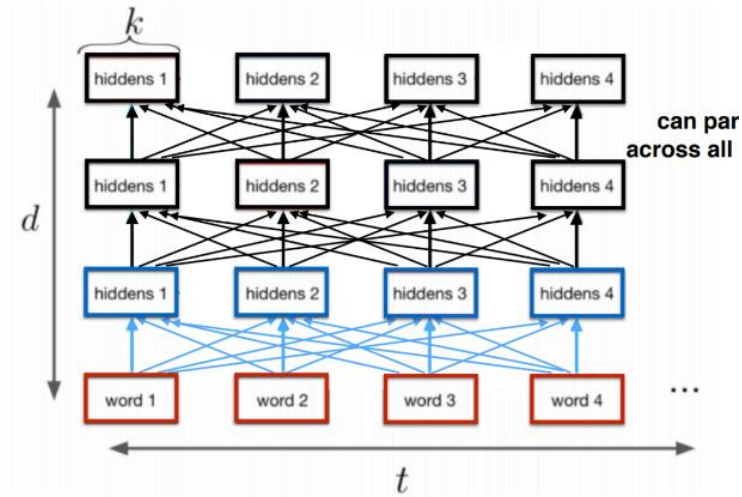
52. Conditional Generation Complexity:
    a. W/O Attention: O(n) for encoding, O(m) for decoding → O(n+m).
    b. W/ Attention: O(n) for encoding, then for each step of the decoder, the attention needs to be computed. That is O(n x m) → O(n x m).
53. Self-Attention: (concept introduced in "Attention is all you need") Self-attention mechanism is a attention applied on an embedded sentence, which generates a new embedding for the word only that this time is a weighted averaged vector made of the embedding of the other words in the sentence.
    To compute the self-attention of an embedded word, a Query (q), Key (k) and Value (v) vectors are computer from the word by multiplying by three matrices (W_q, W_k, W_v). Then a score is computed based on scaled (divided by square root of the dimension of k) dot product of q and k, the scale helps for stabilizing the gradients. The score indicates how much weight should be given to other words in the sentence. The result is passed through a softmax operation which normalizes the scores (that is the weights) to be positive and sum to 1. The new encoding of the word is computed by the sum of the weighted vectors.
    $$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{n}})\mathbf{V}$$
54. Multi-Headed Attention: This approach adds to the self-attention approach by having multiple q and k representations by multiplying each word by multiple matrices. This improves the ability of the model to focus on different positions. In addition, this gives a better representation of the word because more subspaces are explored and the best is learned. Note that in the end of this process we are left with n vectors, each represent the weightings differently. To map those multiple representations, the outputs are concatenated and multiplied by another matrix to get the final single representation. In order to insert the notion of order, a positional vector is added to the word embeddings on the first layer.

55. **Transformers**: The transformer is composed of an encoder and decoder parts, where the encoder is composed of 6 encoding layer and the decoder is composed of 6 layers of decoders.
    The each encoding later is made up of a self-attention mechanism and a feed forward NN. The decoder is of the same structure only it has an encoder-decoder attention mechanism between the self-attention layer and the NN. The encoder and decoder layers are connected by residual connections and layer normalization.
    The decoder's self-attention can only take the previously predicted words and not the future ones (set the vectors to -inf), in addition, it creates the Query matrix from the layer below however the Key and Value matrices are taken from the encoders output.
    The network is finally passed through a linear layer and a softmax.
    Pros: Transformers can be computed in parallel. In addition, information can flow very easily from any point in the sequence to any other point.



56. **Parallelization**:
    RNN case: in the basic case, the running time is proportional to $t * d * k$, however it can be optimized by running the diagonals in parallel to get the time complexity to be proportional to t.
    Self-Attention: Here the layers can be computed in parallel and therefore the time complexity is proportional only to d.





57. Masked Language Model: The idea of masked language model is to hide some tokens in the sequence (and randomly scramble some others) and train the encoders on the task of predicting those tokens. This gives us a language model, where each toke has an encoded representation based on it's context.
58. Inverse Problems: This kind of problems are such that the data generation is easy, and therefore one can create a very large dataset to train on easily. In machine learning we are interested to predict an output base on an input, that is we want to learn F such that $y = F(x)$. In many cases we have the expected result y, but not the input x. However, in inverse problems we can easily generate many x's based on known y's to create training that set. That is in some cases we know F^{-1} and therefore we can calculate $x = F^{-1}(y)$. One problem includes the problem of image-deblurring, where we have many 'cleaned', original images. We can easily apply blurring on the original image to get many pair of cleaned-blurred images to train on.
59. Transfer Learning: When a pre-trained model is used to some other, 'bigger' problem.
60. Universal Language Model (UML): A language model is considered universal if
    a. It works with no dependence on the document/sequence size, and the number and type of the labels.
    b. One architecture to train.
    c. No pre-processing / feature engineering is necessary.
    d. No data other than the data used for the specific task is needed.
61. ULM-FiT: LSTM (3-layers) based architecture implementing a language model. The special part was that this architecture introduced an effective approach to fine-tune the pre-trained model when used as an encoder to solve other problem.
    The way this model works, it by training the LSTM layers on a general purpose corpus to become task independent. This trained model is then used in some other task where the model is fine-tuned by using discriminative fine tuning and slanted triangular learning rates (was not discussed in class). The fine tuning use gradual unfreezing approach where only the last later of the model is unfreezed, unfreezing each consecutive layer after each epoch.

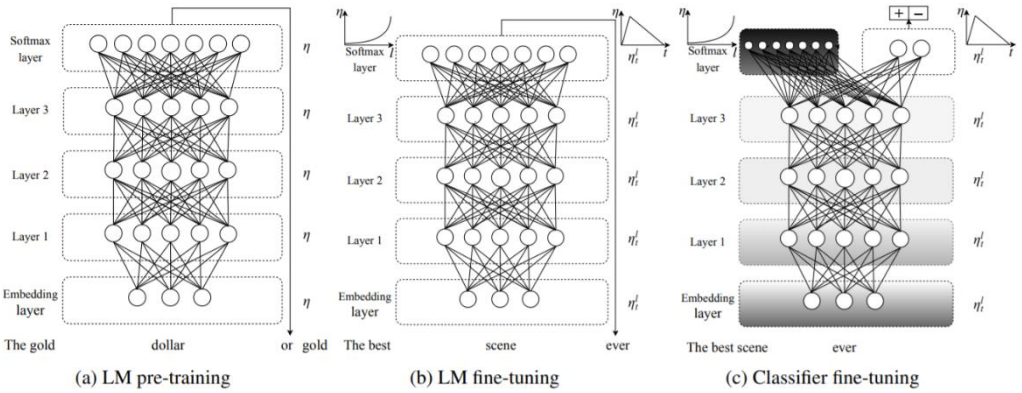
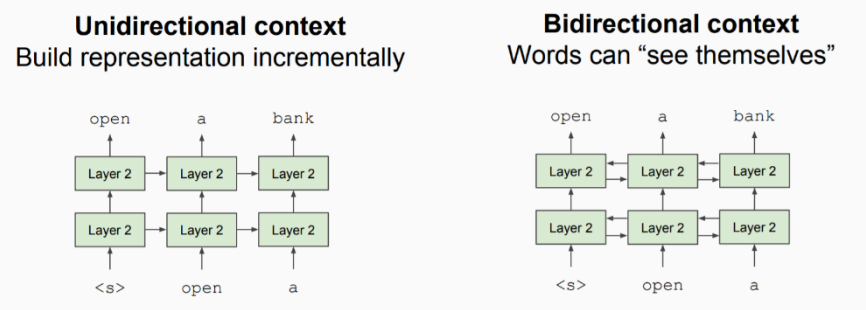


Figure 1: ULMFiT consists of three stages: a) The LM is trained on a general-domain corpus to capture general features of the language in different layers. b) The full LM is fine-tuned on target task data using discriminative fine-tuning ('*Discr*') and slanted triangular learning rates (STLR) to learn task-specific features. c) The classifier is fine-tuned on the target task using gradual unfreezing, '*Discr*', and STLR to preserve low-level representations and adapt high-level ones (shaded: unfreezing stages; black: frozen).

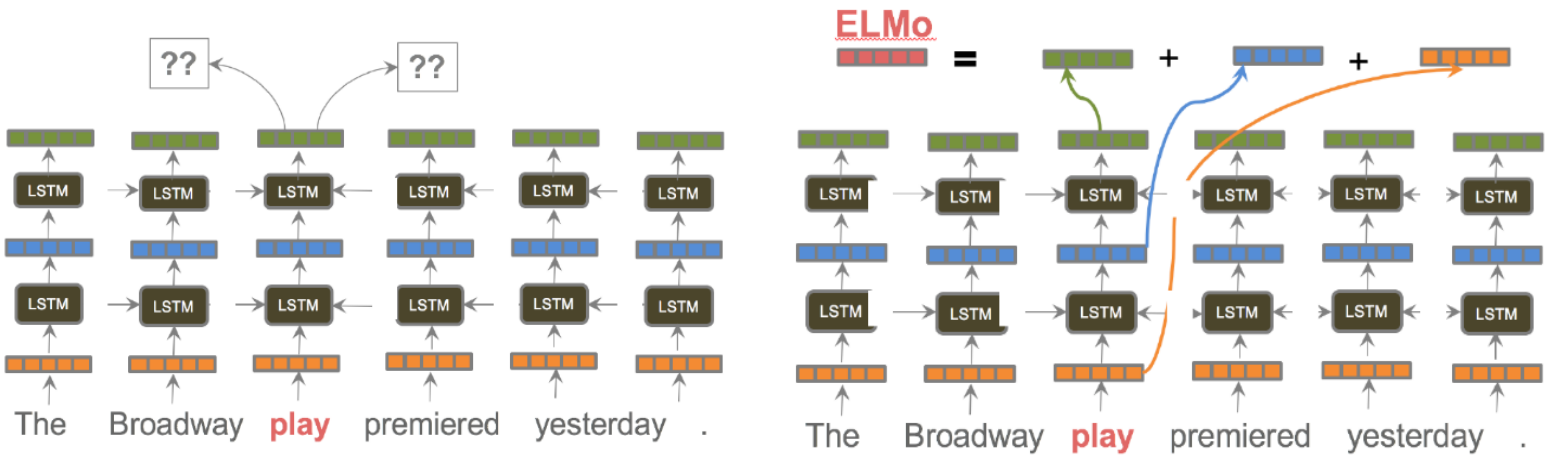
$$LSTM(\mathbf{x_{1:n}}) = \mathbf{h_1}, ..., \mathbf{h_n} = \mathbf{H}$$
$$\tilde{\mathbf{h}} = [\mathbf{h_n}, \text{maxpool}(\mathbf{H}), \text{avgpool}(\mathbf{H})]$$
$$\hat{y} = softmax(MLP(\tilde{\mathbf{h}}))$$

62. OpenAI: This model is based solely on the decoder of a transformer. This is useful because the decoder is built to predict future tokens. This architecture still uses a self-attention layer, however the future tokens are masked. This again, can be used as a pre-trained network in a bigger task.

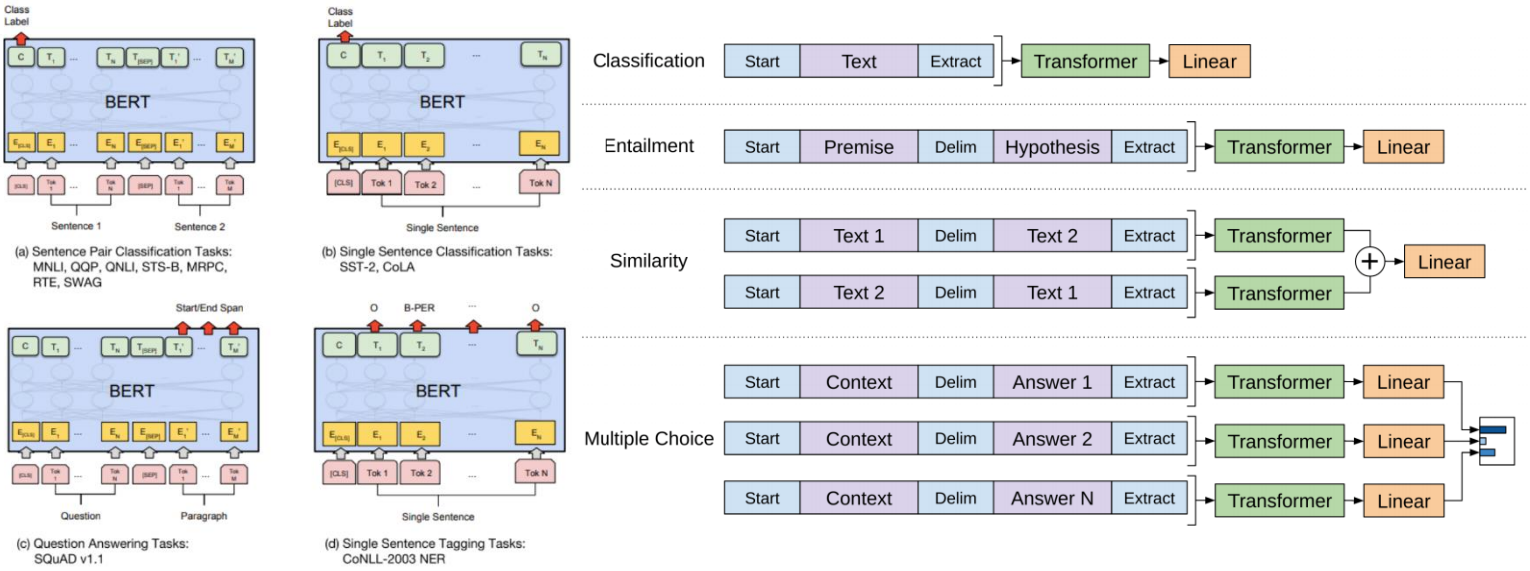
63. ELMo: this network introduced a new way of computing word embeddings. ELMo generates a vector representation based on the context context. Therefore, one can get multiple



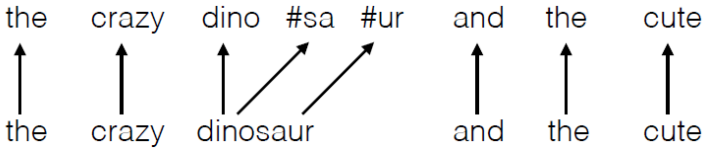
representations of the same token based on different contexts. This is different from the traditional word embedding approach where a constant vector embedding is trained for each token and is used absolutely. The embeddings are created using a deep Bi-LSTM network. ELMo embeddings are created by trying to predict the next word in the sequence in both directions. The output of ELMo is a weigh ted sum of the concatenated outputs from all layers. Note that the authors recommend on concatenating the context-independent word embeddings (like GloVe) to the output of the network. In addition, this model share the forward and backward task in such a way that both are the same final token (here 'play'). The problem with the bidirectionality in this architecture is that the word 'sees' itself, in other word the network can learn to save the next word in the previous layer and pass it to the next one, eventually always guessing correctly. How does ELMO's approach solves this problems?



64. **BERT**: Bert is an <u>encoder only</u> attention network with deep sacked encoding layers. The model is usually used as a pre-trained network in some semi-supervised / supervised classification tasks. Similarly to vanilla attention, BERT accepts a sequence of words (that starts with [CLS] symbol), and for each layer the sequence is passed through a self-attention mechanism and the results are pass through a feed-forward network to the next layer. The output of the BERT encoder is then passed through a classification layer to perform the given task. BERT uses the masked attention model and masks 15% where 80% of those tokens are replaces with $[MASK]$ token, 10% are not changed and 10% are switched with a random token. The model is asked to predict all those values. The latter 20% are helpful for test time and fine-tuning, since the $[MASK]$ token is not used after pre-training. The trade-off of masking is that if two few words are masked, then it is difficult to train. However, if you mask too much, then there



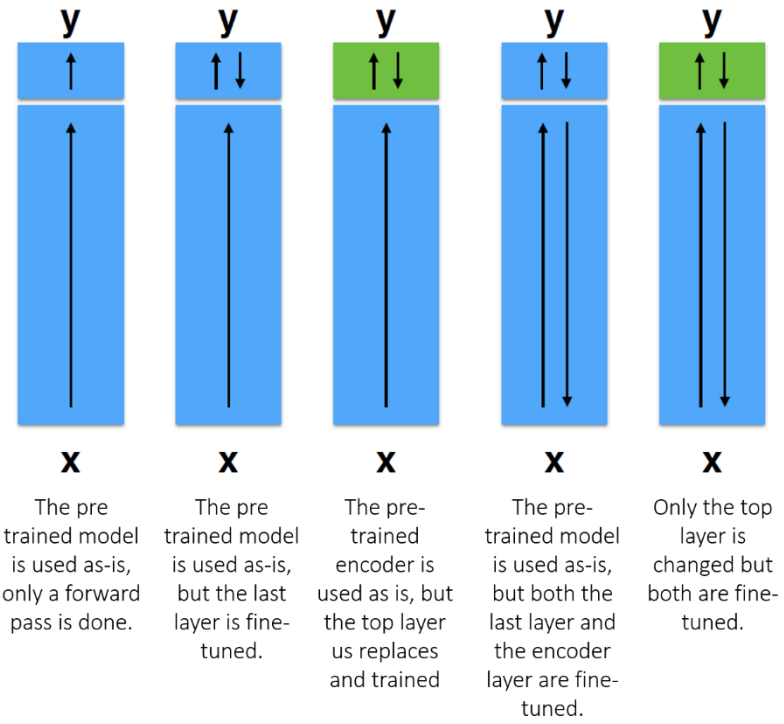
is not enough context for the words.
Another issue in language models is that the vocabulary is too big. This can be solved by breaking the tokens into 'word pieces' where each word is broken down deterministically.



Alternatively, this can be solved by using characters as tokens (or even bytes).

Bert is computationally expensive because the operations are difficult for parallelization. In addition we use it on very long sequences for predicting a single masked token. This rises the question of how we can make those models more efficient and less costly. Research shows ('Green AI') how costly running a

65. Using pre-trained networks:



66. Knowledge Distillation: a method used to boost BERT's running time. The idea behind is to train a smaller 'student' model that learns to mimic the larger 'teacher' model. The distillation model uses the representations outputted from the last layer, however before it had passes the softmax layer. The idea behind distillation lays in this approach. What it is trying to do is to prevent the from being too sure of its prediction. The softmax layer outputs distribution where most of the classes get a probability close to zero, which are ignored in the softmax. However, those values are very different from each other and therefore they store important information. Now to mimic the 'teacher' network, the 'student' network needs to learn the output distributions of the larger network so it can generalize in the same way. To do this we set a different loss function. First it is based on the soft prediction (before the sofmax) secondly it is defined as follows, where t is the teacher's logits and s is a student's logits:

$$L = -\sum_i t_i * log(s_i)$$

Using this loss function, a single example enforces more constraint. In addition, 'soft temperature' is used the probability distribution. Can be used to teach the smaller model to mimic the larger model.