Luis Fred
luisfred.com.br

Deep Learning

Uma introdução ao deep learning com foco em NLP

REGULARIZAÇÃO

Como evitar o overfitting

Regularização

Ter um modelo capaz de reduzir os erros de predição ao mínimo tanto no conjunto de aprendizagem quanto em dados inéditos pode ser algo desafiador. A **regularização** é um conjunto de estratégias adotadas com o objetivo de diminuir a taxa de erros no conjunto de testes, na esperança de obter modelos com melhor capacidade de generalização. Trata-se de um problema antigo, que movimenta toda uma área de pesquisa em torno do assunto. A maioria das abordagens de regularização consistem em penalizar a função de perda por meio da inclusão de um termo $\Omega(\mathbf{w})$ que controla o tamanho dos pesos durante durante a retropropagação.

Regularização

Como o termo *bias* possui menos importância no processo de aprendizagem da rede neural do que os pesos w, ele normalmente não é regularizado.

Regularização

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \alpha\Omega(\mathbf{w})$$

A contribuição relativa de $\Omega(\mathbf{w})$ na função de perda não regularizada é ponderada por um $\alpha \in [0, \infty)$. Quando $\alpha = 0$, não há regularização. O tamanho da regularização é, portanto, proporcional ao valor de α.

Evitando o overfitting

Em deep learning, há algumas técnicas que você pode adotar para evitar overfitting e combinar com outras estratégias que não são regularização exatamente, mas que ajudam. Para citar algumas bastante populares:

- L^2
- Dropout
- Early Stopping (não é regularização)

L^2

A norma L^2 é uma estratégia de regularização bastante comum e é uma das mais simples, sendo amplamente conhecida como **decaimento dos pesos** no contexto das redes neurais.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

A L^2 orienta os pesos a se manterem sempre mais próximos do seu valor inicial, com α governando a importância relativa do termo de regularização na função objetivo não regularizada $J(\mathbf{w})$.

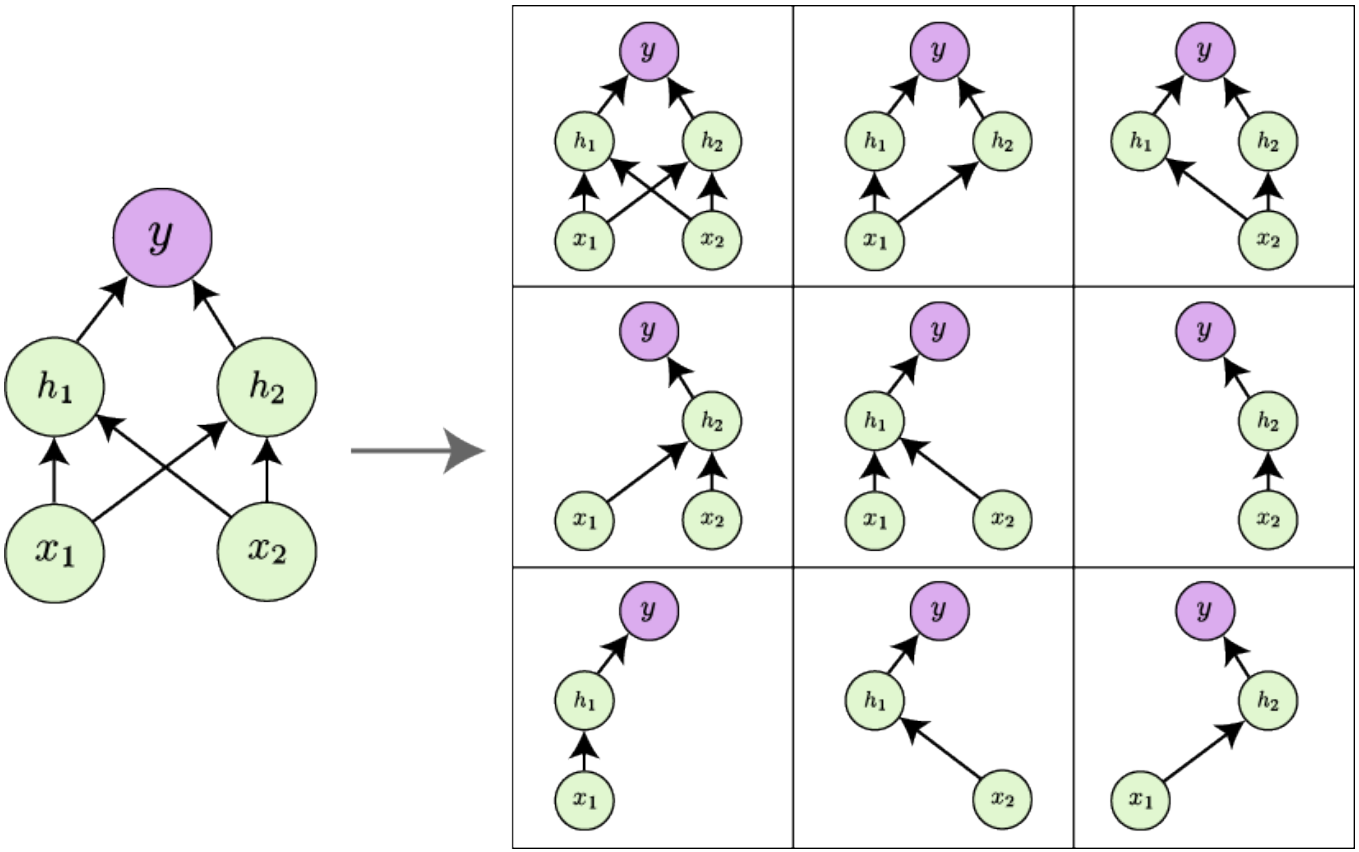
Dropout

O **Dropout** é um método de regularização altamente eficiente, que se tornou um dos mais populares na comunidade de deep learning. Em alguns casos, é possível obter cerca de 2% a mais de acurácia, mesmo em modelos no estado da arte, apenas adicionando o Dropout (não parece pouco se você já tem um modelo com 95% de acurácia). A ideia por trás do dropout é treinar um ensemble de todas as redes neurais que podem ser obtidas ao longo de todas as iterações, se a cada iteração nós desconectarmos alguns neurônios da rede base. Cada neurônio tem um probabilidade p de ser desconectado (*dropado*).

Dropout

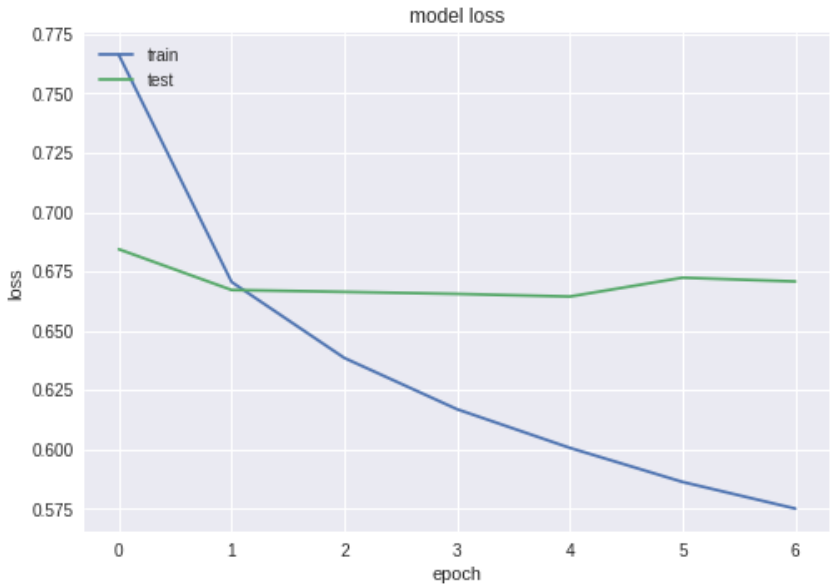
Deste modo, a cada iteração, uma rede neural diferente, com uma quantidade diferente de unidades ativas, é obtida. Isso acontece porque cada neurônio pode está ou presente ou ausente em cada iteração. Sendo N o número total de neurônios que podem ser desconectados, no final do processo é como se tivéssemos um ensemble de 2^N redes neurais diferentes compartilhando seus pesos. A rede neural resultante pode ser vista como uma média de todas essas redes neurais menores que foram obtidas.

Dropout



Early Stopping

Ao treinar modelos de deeplearning é comum vermos o valor do erro diminuir no conjunto de aprendizagem e aumentar no conjunto de testes ou validação a partir de um determinado ponto.



Early Stopping

Isso significa que nós conseguiremos obter um modelo com melhor capacidade de generalização se interrompermos o treino exatamente neste ponto, onde o erro no conjunto de testes ou validação passa a aumentar. Quando o treino termina, nós ficamos com os valores dos pesos ajustados nesse ponto, ao invés dos pesos mais recentes que não seriam os melhores. Para isso, nós precisamos criar um checkpoint com os parâmetros do modelo a cada vez que o erro diminui e usamos estes parâmetros depois que o treino terminar. Trata-se de uma outra maneira de incluir regularização em modelos de aprendizagem de máquina e normalmente chamado de *Early Stopping*.

Batch Normalization

Batch Normalization

Otimizadores baseados no *Gradiente Descendente Estocástico* exigem uma cautelosa escolha dos hiperparâmetros do modelo, sobretudo da taxa de aprendizagem, e também um maior rigor na inicialização dos pesos. Além disso, treinar uma rede neural pode ser um processo complicado pelo fato de que as constantes mudanças nos parâmetros das primeiras camadas também afetam as entradas das camadas seguintes e a distribuição das ativações. Desse modo, a mudança na distribuição das entradas ao longo da rede pode trazer prejuizos para o ajuste do modelo porque as outras camadas precisam se adaptar o tempo inteiro a essas mudanças de distribuição.

Batch Normalization

Então, é mais vantajoso que **a distribuição das entradas seja mantidas fixa ao longo do treino**, para que os parâmetros das camadas à frente não tenham que passar por reajustes com o objetivo de compensar alterações nesta distribuição.

Batch Normalization

A aplicação de uma normalização nos parâmetros, para cada *mini-bach*, no sentido de manter constantes as médias e as variâncias das camadas de entrada é a grande sacada do **Batch Normalization** para contornar esses problemas e, assim, aceletar o treino da rede. Com o BN, há também um efeito benéfico no fluxo do gradiente, ao reduzir a necessidade de manter os valores dos gradientes na mesma escala dos valores iniciais dos pesos, além de permitir o uso de taxas de aprendizagem maiores. Em alguns casos, o BN também pode atuar como regularizador, dispensando a necessidade de Dropout.

Batch Normalization

Seja $\boldsymbol{H}$ um mini-bach contendo as ativações de uma camada que precisa ser normalizada. Para normalizar a matriz $\boldsymbol{H}$, ela precisa ser substituída por:

$$\boldsymbol{H'} = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

onde $\boldsymbol{\mu}$ é um vetor contendo as médias das ativações das unidades (neurônios) para cada mini-batch H, e $\boldsymbol{\sigma}$ é um vetor contendo os respectivos valores de desvio padrão. A normalização é aplicada a cada linha de $\boldsymbol{H}$, produzindo $\boldsymbol{H'}$ que, então, é usada pela rede rede.

Recurrent Neural Networks - RNN

Quando a ordem das features é importante

RNN

RNN's são um tipo de rede neural especialmente concebidas para lidar com dados sequenciais do tipo $x_i, ..., x_j$. Elas conseguem lidar com sequências muito mais longas do que as redes MLP convencionais, graças ao compartilhamento de parâmetros que acontece entre diferentes partes do modelo. O compartilhamento de parâmetros é uma habilidade importante nas RNN's, considerando que uma determinada informacão pode se repetir em diferentes posições em uma sequência. Por exemplo, uma palavra que pode aparecer em mais de um local dentro de uma sentença e, ao mesmo tempo, ser importante para o significado do texto. Para conseguir incorporar o contexto de uma sequência a cada passo de tempo, uma memória dos passos de tempo anteriores precisa ser preservada.

RNN

Seja $\boldsymbol{T}$ o tamanho de uma sequência de entrada $\boldsymbol{X}$, onde $\boldsymbol{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_T}\}$, tal que $\boldsymbol{x_t} \in \mathbb{R}^N$ seja um vetor de entrada num tempo t. Nós definimos a memória, ou histórico, até o tempo t como $\boldsymbol{h_t}$. Assim a saída $\boldsymbol{o_t}$ é definida como sendo:

$$\boldsymbol{o_t} = f(\boldsymbol{x_t}, \boldsymbol{h_{t-1}})$$

onde a função f é responsável por mapear a entrada $\boldsymbol{x_t}$ e a memória $\boldsymbol{h_{t-1}}$ para uma saída. Assim, a memória num passo de tempo t é $\boldsymbol{h_{t-1}}$ e a entrada é $\boldsymbol{x_t}$. A saída $\boldsymbol{o_t}$ pode ser considerada como um vetor contendo o histórico da sequência inteira até o passo t. Assim sendo: $\boldsymbol{h_t} = \boldsymbol{o_t} = f(\boldsymbol{x_t}, \boldsymbol{h_{t-1}})$.

RNN

O termo "recorrente" vem do fato de que a mesma função f é usada para cada instância, onde a saída é diretamente dependente dos resultados anteriores.

RNN

A tranformação aplicada pela função f é mais formalmente definida da sequinte forma:

$$\boldsymbol{h_t} = f(\boldsymbol{U x_t} + \boldsymbol{W h_{t-1}}) + \boldsymbol{b}$$

onde $\boldsymbol{W}$ e $\boldsymbol{U}$ são matrizes de pesos com $\boldsymbol{W}, \boldsymbol{U} \in \mathbb{R}^{(NxN)}$ e f é uma função de ativação, como a ReLU. As RNN's são treinadas com *retropropagação* e gradiente descendente, da mesma forma que a rede neural feed-foward que vimos antes.

RNN

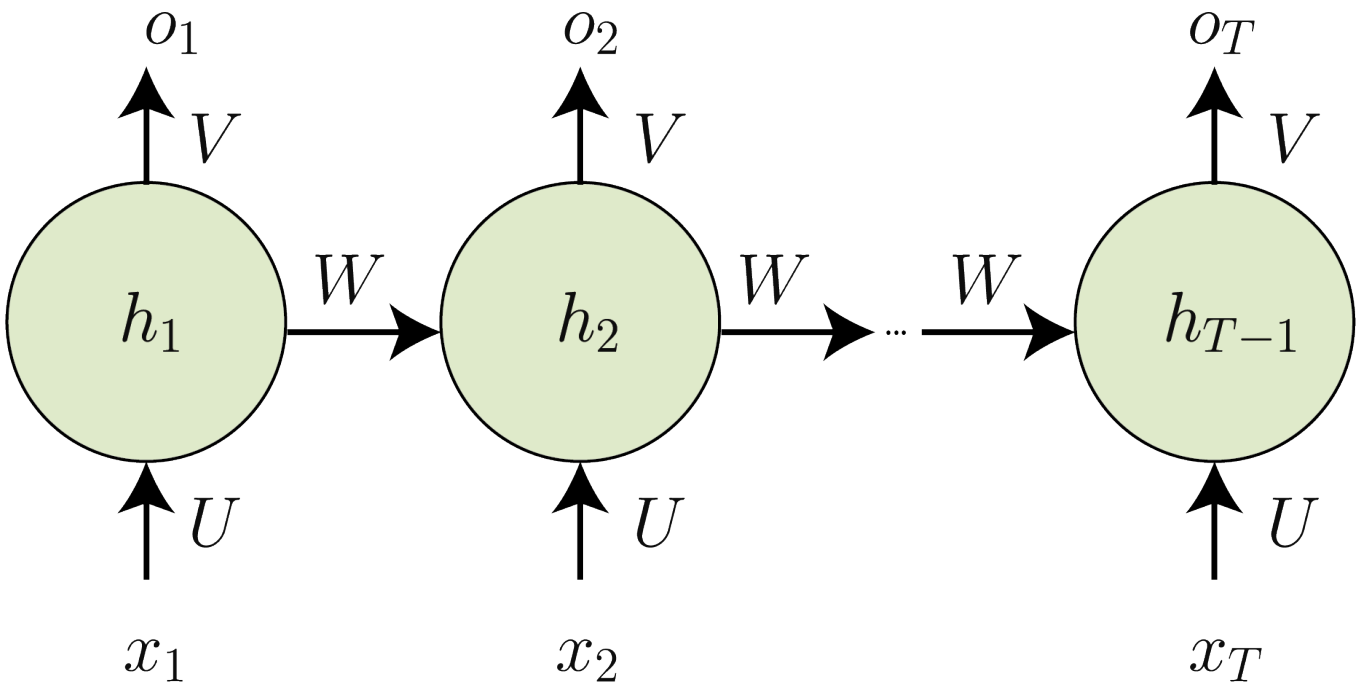
Em cada passo de tempo t, um vetor de saída $\hat{\boldsymbol{y}}_{\boldsymbol{t}}$ é obtido da seguinte forma:

$$\boldsymbol{h_t} = \tanh(\boldsymbol{U x_t} + \boldsymbol{W h_{t-1}}) + \boldsymbol{b}$$

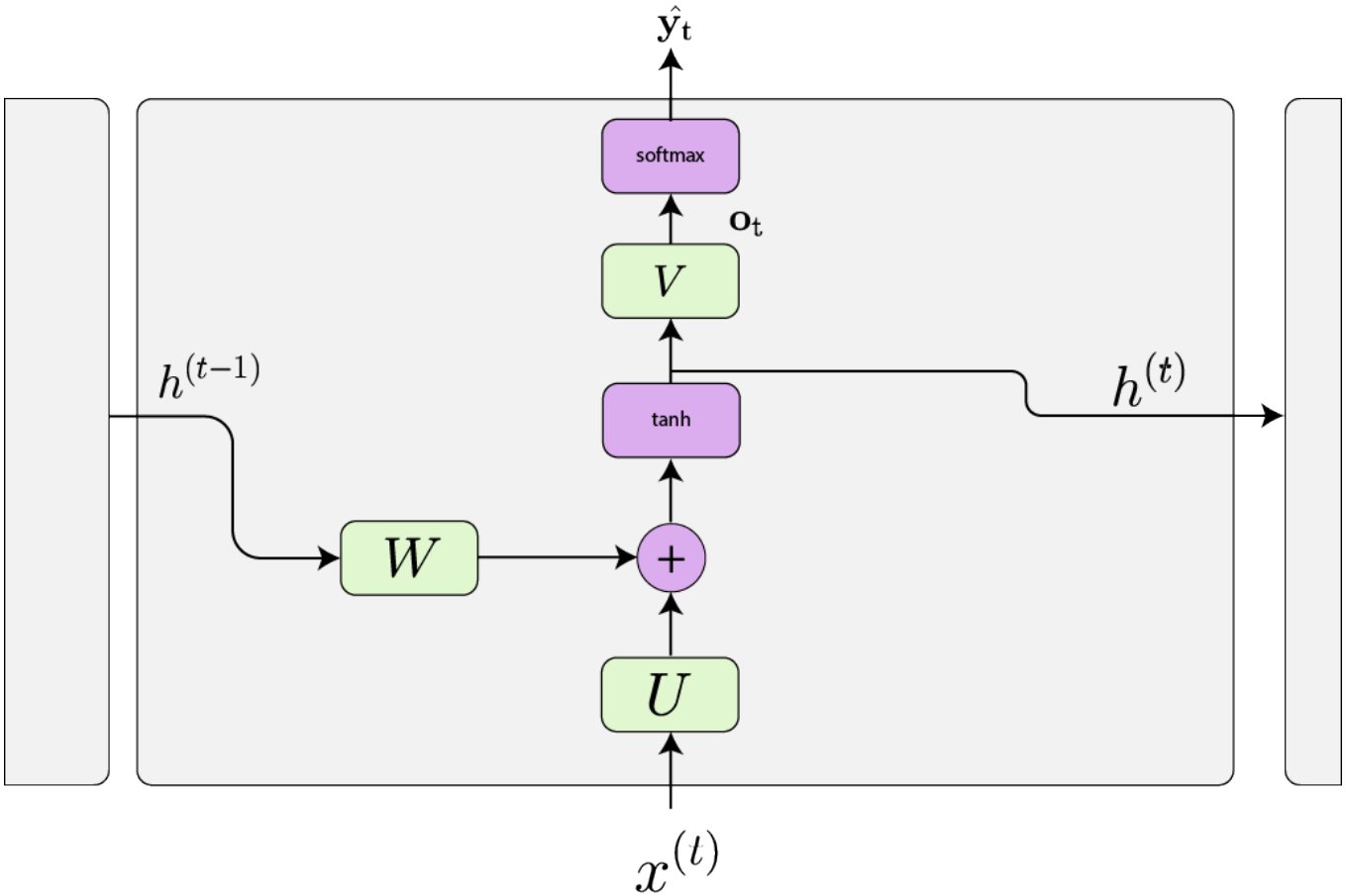
$$\boldsymbol{o_t} = \boldsymbol{V h_t}$$
$$\hat{\boldsymbol{y}}_{\boldsymbol{t}} = softmax(\boldsymbol{o_t})$$

onde os parâmetros a serem treinados são $\boldsymbol{U}$, $\boldsymbol{W}$ e $\boldsymbol{V}$. $\boldsymbol{U}$ incorpora a informação de $\boldsymbol{x_t}$, $\boldsymbol{W}$ incorpora o estado oculto recorrente e $\boldsymbol{V}$ é responsável por parametrizar a saída e a classificação. Assim, o gradiente da função de perda deve ser computado com respeito a estas matrizes, durante a retropropagação.

RNN



RNN



O problema da dissipação do gradiente

dissipação do gradiente

Um dos aspectos das RNN's que tornam o treino delas muito mais difícil é o problema da **dissipação do gradiente**, frequentemente referenciado como **vanishing gradient** na literatura. Durante a retropropagação, as sucessivas operações matemáticas envolvendo os gradientes ao longo dos vários estágios diminuem exponencialmente as contribuições dos pesos. Isso faz com que os sinais das primeiras entradas percam influência (devidos à pesos muito pequenos) à medida em que se tornam mais distantes. Isso afeta muito negativamente a performance da RNN ao lidar com sequências maiores.

Dissipação de gradientes

Redes recorrentes com mecanismos de controle de fluxo permitem que a rede acumule informação sobre as features por um longo período. Para isso, elas contam com funções que controlam os estados da sequência e determinam quanta informação é mantida na memória e qual parcela da informação é propagada adiante. Isso não é definido de maneira estática, pois a rede neural precisa aprender sobre as condições necessárias para a tomada deste tipo de decisão. Há duas dessas arquiteturas baseadas em mecanismos de controle de acesso à memória que são amplamente conhecidas. São elas: **LSTM** - *Long Short-Term Memory* e a **GRU** - *Gated Recurrent Units*, são uma resposta para esse problema.

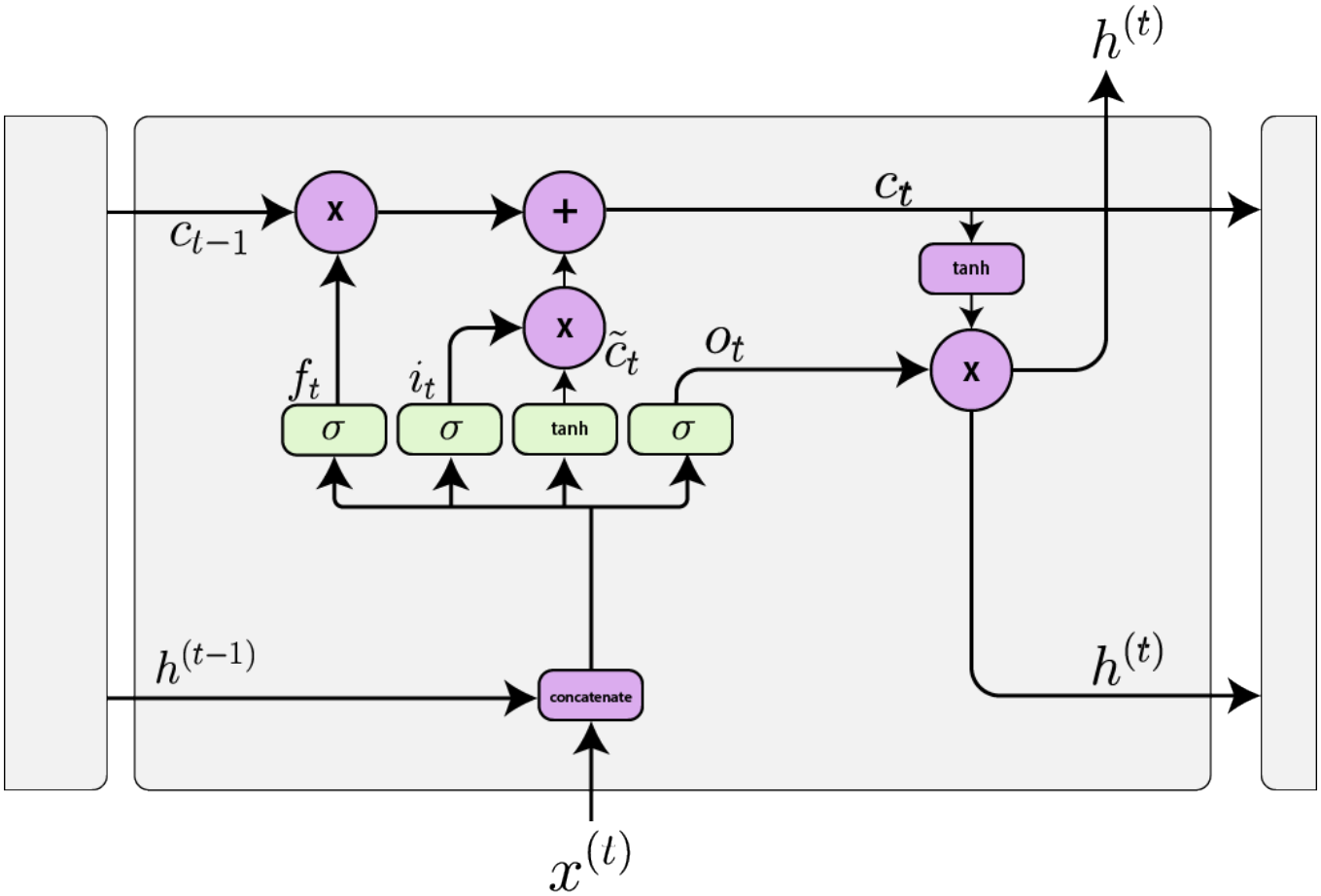
Long Short-Term Memory - LSTM

LSTM

As redes LSTM se valem de mecanismos de controle frequentemente referenciados como *gates* na literatura. Esses mecanismos controlam a maneira como o gradiente se propaga pela memória da RNN e protegem a célula de memória que está conduzindo o histórico para as células seguintes. Esses *gates* são, na verdade, camadas de redes neurais mais simples. Isto permite que a RNN aprenda as condições necessárias para ignorar ou manter informações na memória. São três *gates*:

- Input gate
- Forget gate
- Output gate

LSTM


$h^{(t)}$
c_t
c_{t-1}
x
+
tanh
x
$\tilde{c}_t$
o_t
f_t
i_t
σ
σ
tanh
σ
x
$h^{(t-1)}$
concatenate
$h^{(t)}$
$x^{(t)}$

LSTM

Uma célula LSTM é formalmente definida assim:

$$
\begin{aligned}
\mathbf{i_t} &= \boldsymbol{\sigma}(\mathbf{W}_i \boldsymbol{x_t} + \mathbf{U}_i \boldsymbol{h_{t-1}} + \boldsymbol{b_i}) \\
\mathbf{f_t} &= \boldsymbol{\sigma}(\mathbf{W}_f \boldsymbol{x_t} + \mathbf{U}_f \boldsymbol{h_{t-1}} + \boldsymbol{b_f}) \\
\mathbf{o_t} &= \boldsymbol{\sigma}(\mathbf{W}_o \boldsymbol{x_t} + \mathbf{U}_o \boldsymbol{h_{t-1}} + \boldsymbol{b_o}) \\
\mathbf{\tilde{c}_t} &= \tanh(\boldsymbol{\sigma}(\mathbf{W}_c \boldsymbol{x_t} + \mathbf{U}_c \boldsymbol{h_{t-1}}) \\
\mathbf{c_t} &= \mathbf{f_t} \circ \mathbf{c_{t-1}} + \mathbf{i_t} \circ \mathbf{\tilde{c}_t} \\
\mathbf{h_t} &= \mathbf{o_t} \circ \tanh(\mathbf{c_t})
\end{aligned}
$$

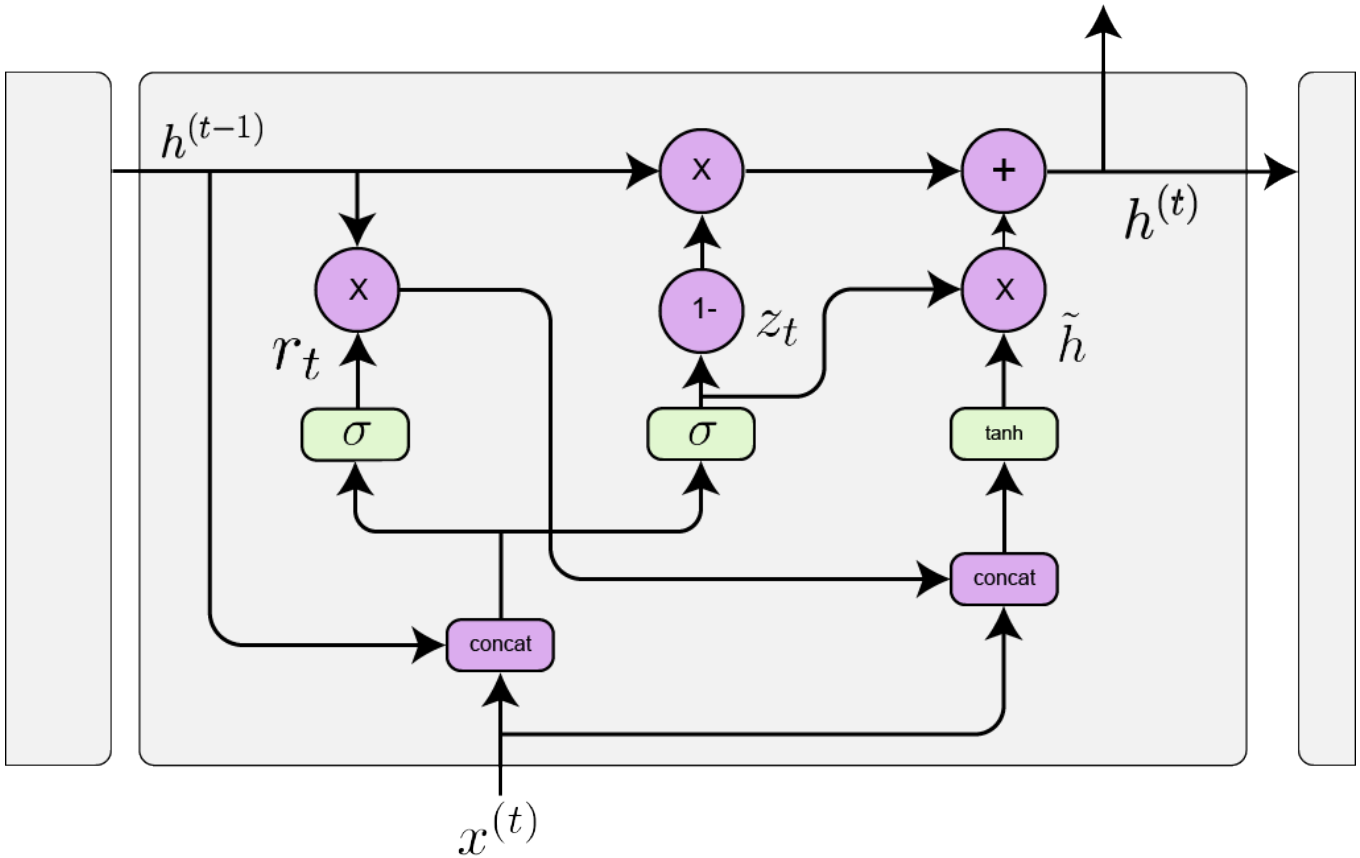
O mecanismo de controle $\mathbf{f_t}$, também chamado de *forget gate*, controla a quantidade de informação que é mantida na memória durante cada passo de tempo.

Gated Recurrent Unit - GRU

GRU

A GRU é outra arquitetura com métodos de controle para RNN bastante popular. Ela é mais simples de computar e computacionalmente mais eficiente do que a LSTM, já que usa menos parâmetros. Apesar disso, a escolha entre usar LSTM ou GRU é puramente empírica e não se pode concluir facilmente que a GRU tenha uma performance de generalização significativamente superior.

GRU



GRU

As equações que controlam a atualização da célula de memória na GRU são:

$$\mathbf{z_t} = \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1})$$
$$\mathbf{r}_t = \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1})$$
$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h \mathbf{h}_{t-1} \circ \mathbf{r}_t)$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \circ \tilde{\mathbf{h}}_t + \mathbf{z}_t * \mathbf{h}_{t-1}$$

Na GRU, o vetor $\tilde{\mathbf{h}}$, candidato a compor o novo estado, é combinado com o estado anterior, com o mecanismo de controle $\mathbf{z}_t$ determinando que parcela do histórico em memória será conduzida para a próxima célula ou o tamanho da influência que $\tilde{\mathbf{h}}$ terá no histórico.

Convolutional Neural Networks - CNN

O que são CNN's e como elas são aplicadas em NLP?

CNN

Redes convolucionais, ou **ConvNets**, são inspiradas no Córtex visual biológico, o qual possui pequenas regiões de células que são sensíveis a áreas específicas do campo visual. Por exemplo, há neurônios em nosso cérebro que só se ativam diante de determinadas partes de uma imagem, como contornos específicos ou algumas bordas, ou linhas que só estão dispostas em orientações específicas. Esse conceito, no qual certos neurônios desempenham tarefas específicas, forma a base das redes convolucionais.

CNN

As ConvNets têm alcançado performances equiparáveis ou superiores à performance humana em tarefas como classificação de imagens e vídeos. Carros autônomos também são outra aplicação dessas arquiteturas.

CNN

Embora sejam amplamente conhecidas pela sua capacidade e performance superior em tarefas de visão computacional, as ConvNets também têm recebido bastante atenção por terem mostrado um desempenho promissor em tarefas de processamento de linguagem natural, como classificação de textos.

CNN

Como elas advêm da área de visão computacional, boa parte da construção de modelos baseados nessas arquiteturas fazem referência à imagens ou matrizes em 2D. As aplicações em linguagem natural, entretanto, exigem que as entradas sejam mapeadas para 1D, já que dados de texto são representados em uma dimensão.

CNN

O nome **Convolucional** vem do fato de que as redes convolucionais usam operações de **convolução** em parte de suas camadas, no lugar de fazer multiplicação entre matrizes, como ocorre nas redes neurais convencionais. Além disso, disso essas arquiteturas também usam o tipo de operação chamada **pooling**, a qual ajuda a reduzir o tamanho espacial dos dados de entrada e, consequentemente, o número de parâmetros que a rede neural precisa computar.

Convolução - Explicando

Uma convolução é uma operação matemática na qual uma função de entrada $x(t)$ é combinada com uma função $h(t)$, resultando numa saída que é uma sobreposição entre $x(t)$ e $h(t)$. A função $h(t)$ é geralmente conhecida como **kernel**, ou **filtro**. Quando a entrada é uma imagem, o filtro tem tamanho mxn. O resultado desta operação é um tensor $\boldsymbol{F}$ conhecido como **mapa de features**. A convolução é representada como $h * x$.

Convolução - Explicando

Matematicamente, para o caso 1D (processamento de linguagem natural) a convolução é representada assim:

$$F(i) = (h * x)(i) = \sum_n h(n)x(i-n)$$

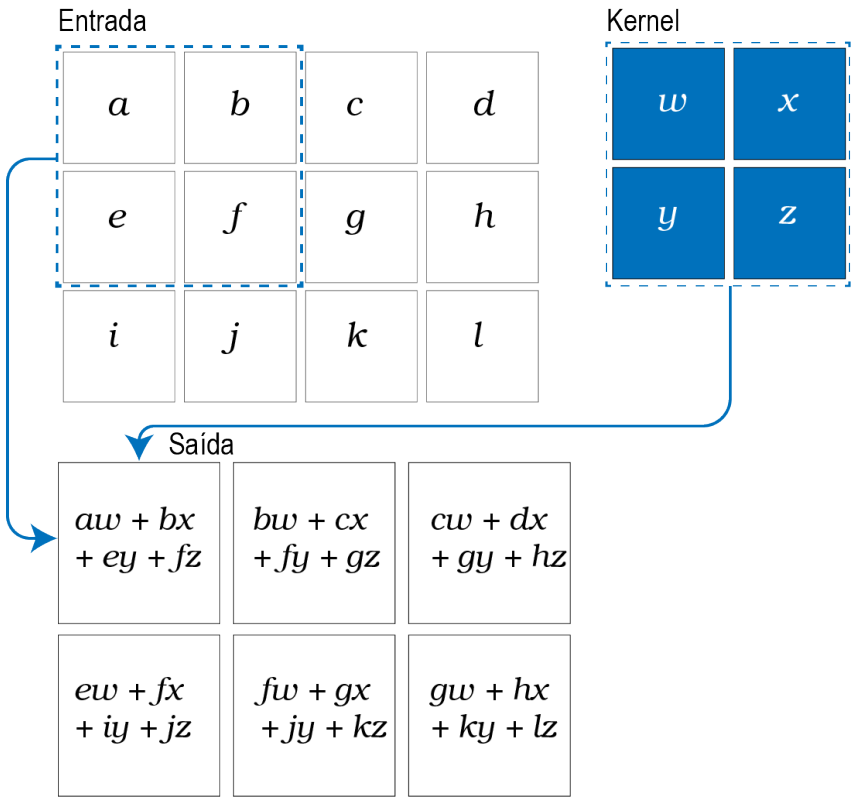
Para o caso 2D, usado em visão computacional (tanto as entradas quanto o kernel são bidimensionais):

$$F(i,j) = (h * x)(i,j) = \sum_m \sum_n x(i+m, j+n)h(m,n)$$

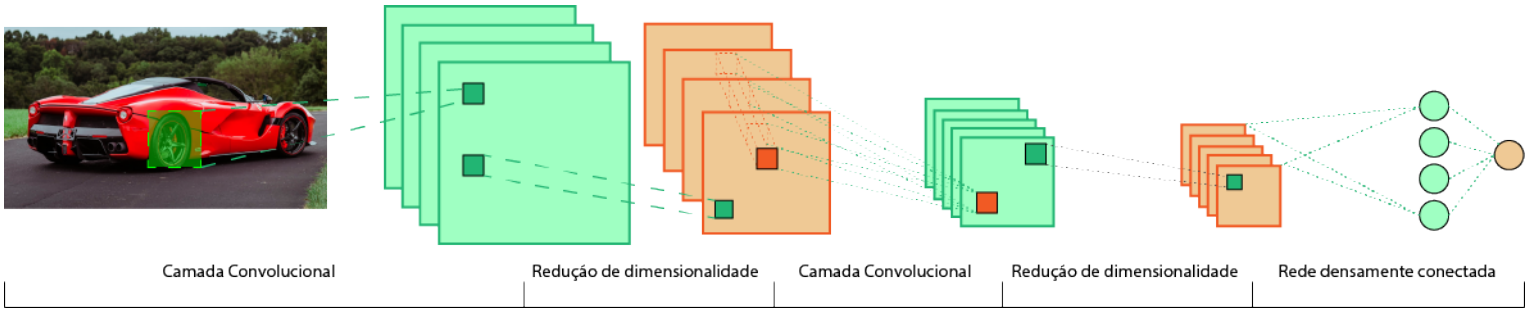
Convolução - Explicando

Seria como deslizar o kernel mxn sobre cada região da entrada, multiplicado as duas matrizes, elemento por elemento, resultando em uma terceira matriz, que é o **mapa de features**.

Convolução - Explicando



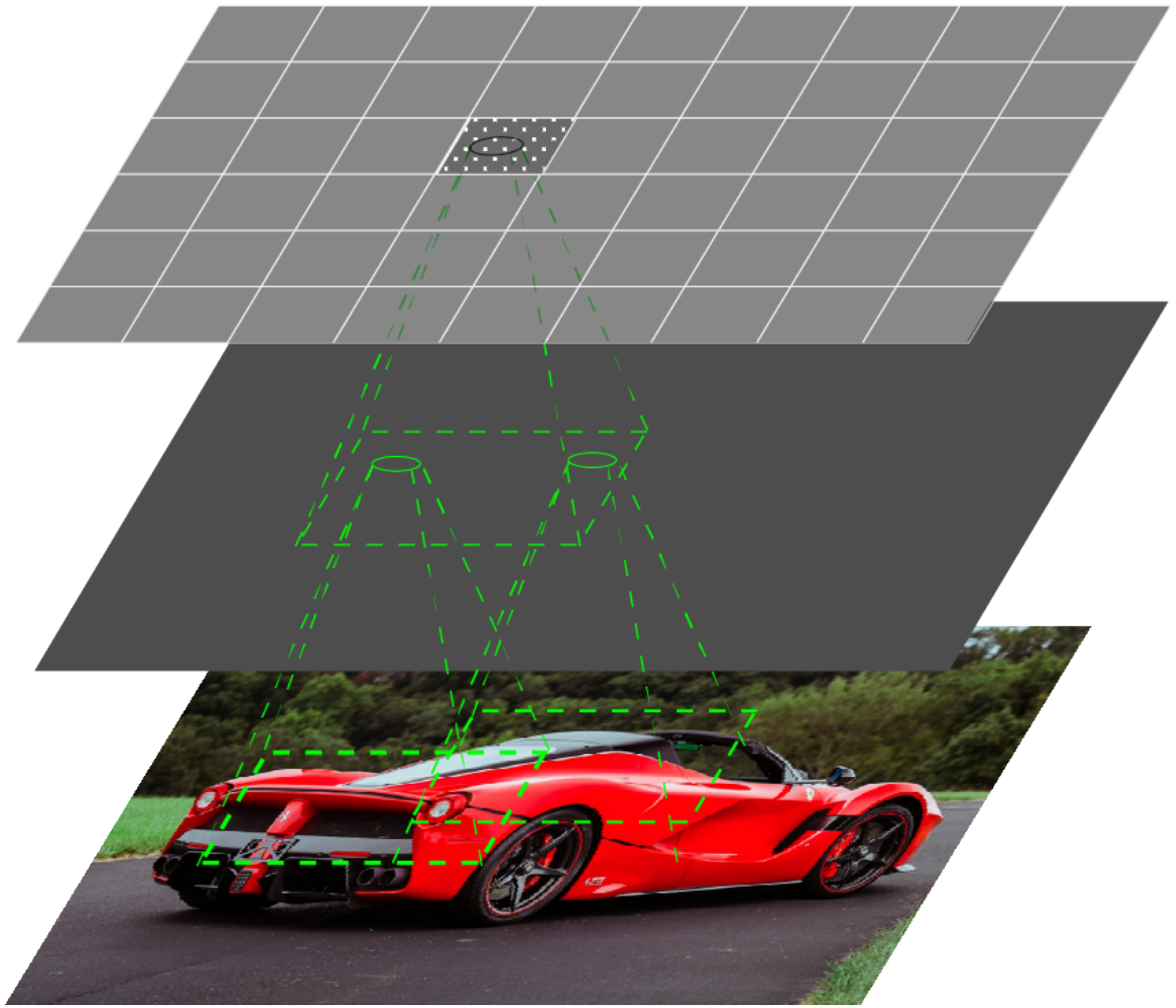
Arquitetura



Camada Convolucional

É o componente mais importante de uma CNN. Os neurônios da primeira camada convolucional não estão conectados a todos os pixels da imagem de entrada de uma vez, mas apenas àqueles pixels que estão dentro do campo de visão do **kernel**. O mesmo acontece com as camadas convolucionais seguintes, se conectando apenas à pequenas sub-regiões da camada imediatamente anterior. Este tipo de comportamento permite que os neurônios features importantes para a representação da imagem, como linhas, contornos, cantos, bordas, etc.

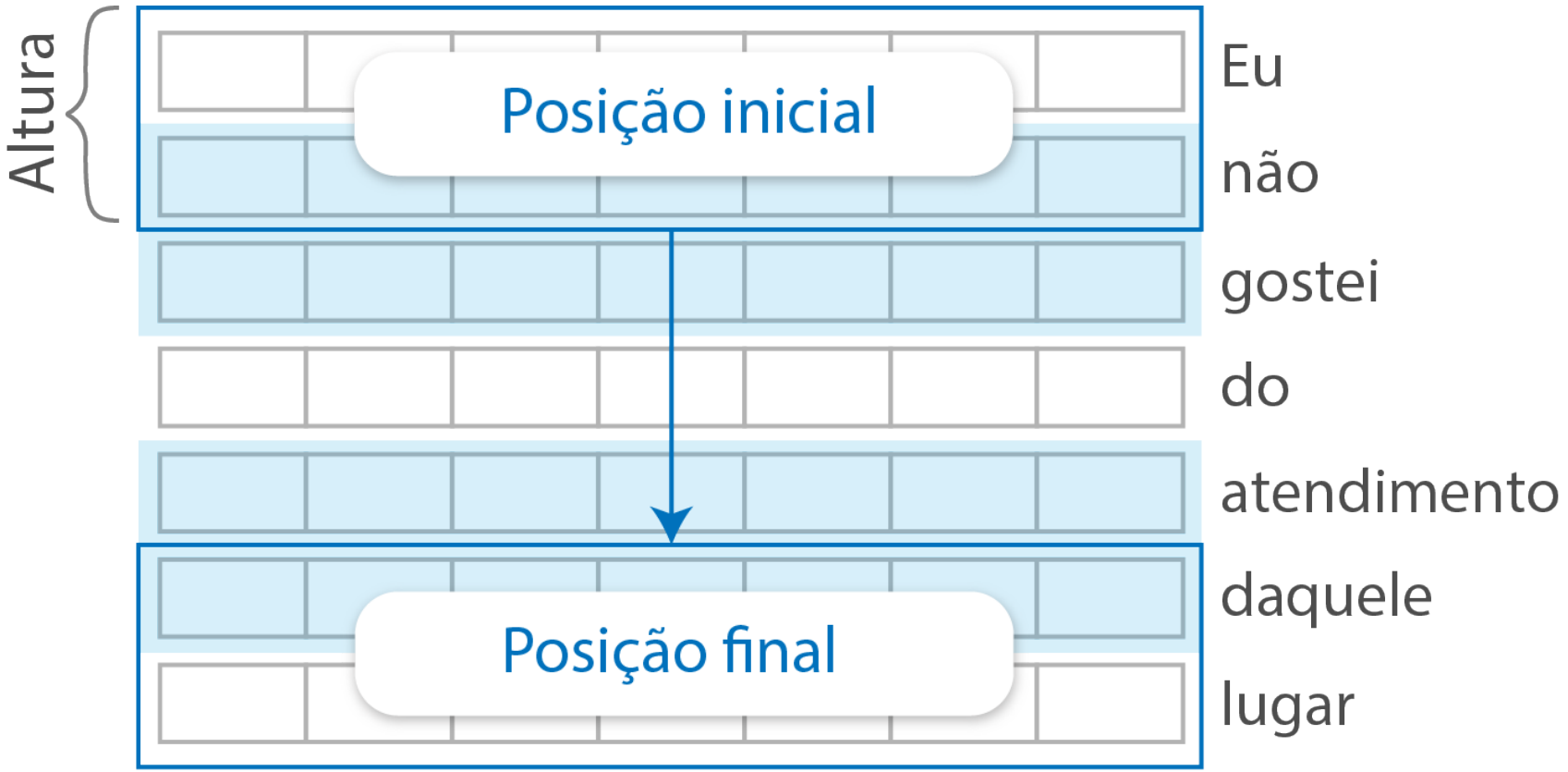
Camada Convolucional



Convolução em dados de texto - 1D

A diferença entre convoluções 1D e outras que ocorrem em mais dimensões está na estrutura dos dados de entrada e como o **kernel** se move em cima desses dados. No caso de dados de texto, cada palavra é representada como um vetor, e o kernel se move em cima desses vetores, escaneando as palavras, só que em apenas uma direção. Pelo menos uma palavra precisa está dentro do campo de visão do kernel.

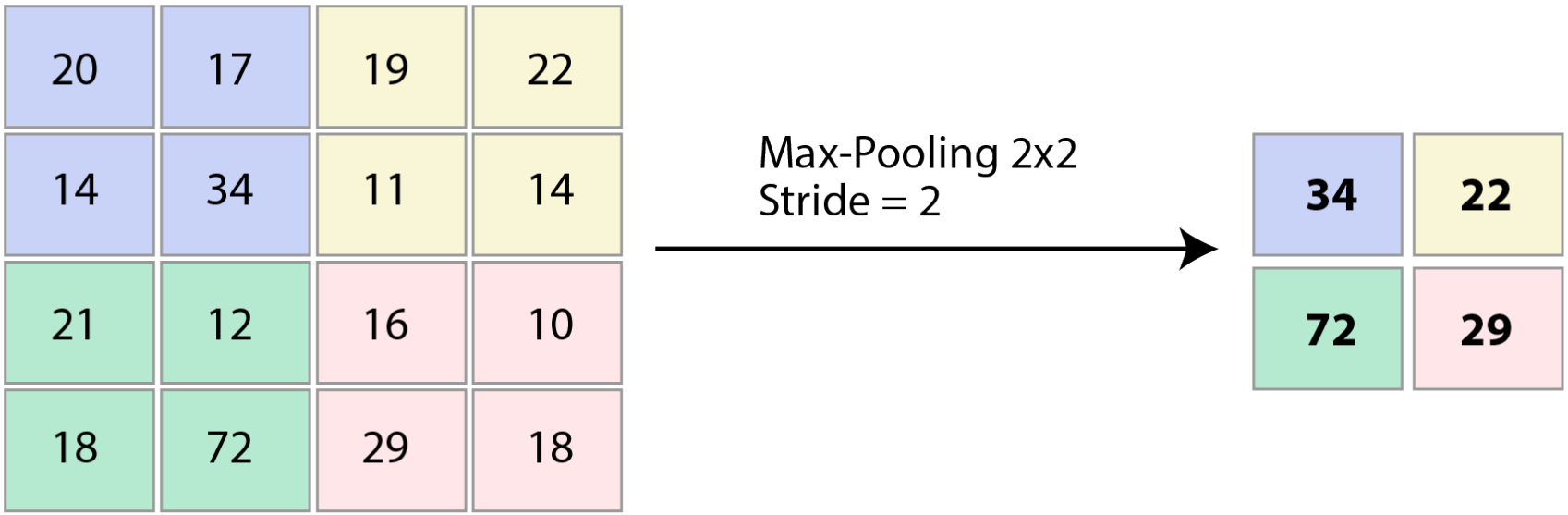
Convolução em dados de texto - 1D



Pooling

O **Pooling** é uma operação que sempre ocorre em cima do resultado de uma camada convolucional e tem o objetivo de reduzir o tamanho espacial da entrada, conservando as informações mais importantes (sim, há perda de informação nesta operação). Desta forma, o pooling consegue reduzir o número de parâmetros da rede, ajudando na performance computacional.

Convolução em dados de texto - 1D



CNN

A estrutura típica de um rede neural convolucional:

- Camada convolucional;
- Função de ativação (ReLU);
- Camada Pooling (redução de dimensionalidade);
- Camada densamente conectada (produz os scores das classes)
- Camada Dropout

Performance

Por que as CNNs têm melhor performance computacional do que as RNNs e as redes Feed-foward?

Performance

Nas redes neurais tradicionais, por conta da multiplicação entre matrizes das entradas e dos parâmetros, todas as unidades em cada camada estão interconectadas e o número de conexões pode se tornar absurdo, dependendo do modelo. Na CNN, o número de conexões é muito inferior porque o tamanho do kernel é sempre muito menor do que o tamanho da matriz de entrada (lembre-se de que o kernel atua em pequenas sub-regiões). Isso significa que a rede armazena menos parâmetros, resultando em menos dados para se ajustarem na memória e menos operações matemáticas para computar a saída da rede.

Word Embeddings

Representação numérica eficiente das palavras de um vocabulário

Word Embeddings

Um Word Embedding é uma matriz de vetores numéricos, onde cada vetor representa uma palavra presente em um vocabulário. Esses vetores codificam várias regularidades semânticas entre as palavras e partem da seguinte ideia: se duas palavras forem usadas em contextos similares, então elas serão associadas à vetores $\mathbb{R}^d$ muito similares. Assim, espera-se que os sinônimos tenham vetores muito similares, tal como outras palavras que são semanticamente muito próximas (Motor e carro). Esse uso da informação semântica das palavras tem aumentado expressivamente a robustês de modelos neurais para NLP.

Word Embeddings

Alguns métodos recentes de representação de palavras usam redes neurais para obter, eficientemente, representações de alta qualidade a partir de grandes conjuntos de textos não estruturados, ao adotar estratégias de aprendizagem não supervisionada para codificar diversas regularidades linguísticas nesses textos.

Word Embeddings

Vocabulário	Vetor			
presidente	-0.093907	0.271813	0.045385	-0.062937
estado	0.254259	0.588025	-0.056152	-0.055573
comissão	-0.306975	0.379277	-0.014631	0.026234

Se tivermos um vocabulário $\mathbf{V} = \{w_1, w_2, ...w_n\}$, com cada palavra sendo mapeada para um vetor $\mathbb{R}^d$, onde d é a dimensionalidade do vetor, então o nosso Word Embedding terá as dimensões $\mathbf{V}$xd.

Word Embeddings

Como as palavras são representadas por vetores, nós podemos usar a medida de **similaridade cosseno** para calcular a similaridade entre essas palavras. Essa medida calcula o ângulo entre os vetores:

$$sim_{cos}(u, v) = \frac{u \cdot v}{\|u\|_2 \|v\|_2} = \frac{\sum_i u_{[i]} \cdot v_{[i]}}{\sqrt{\sum_i (u_{[i]})^2} \sqrt{\sum_i (v_{[i]})^2}}$$

Quanto mais próximo de $\mathbf{1}$ for o valor de $sim_{cos}(u, v)$, mais similares são as palavras que os vetores u e v representam.

Word Embeddings

No lugar de inicializar os pesos da rede aleatoriamente, nós inicializamos usando os valores presentes na matriz Word Embedding. Esta simples mudança geralmente aumenta a performance do modelo substancialmente.

Alguns métodos comuns

- Word2vec
- FastText (considera sub-estruturas das palavras)

Word2Vec

Word2vec

Um dos algoritmos mais populares para a criação de word embeddings é o Word2Vec. Ele usa redes neurais para obter representações de palavras e implementa dois métodos diferentes para representar informações de contexto: **CBOW** e **Skip-Gram**.

Word2vec

CBOW x Skip-Gram

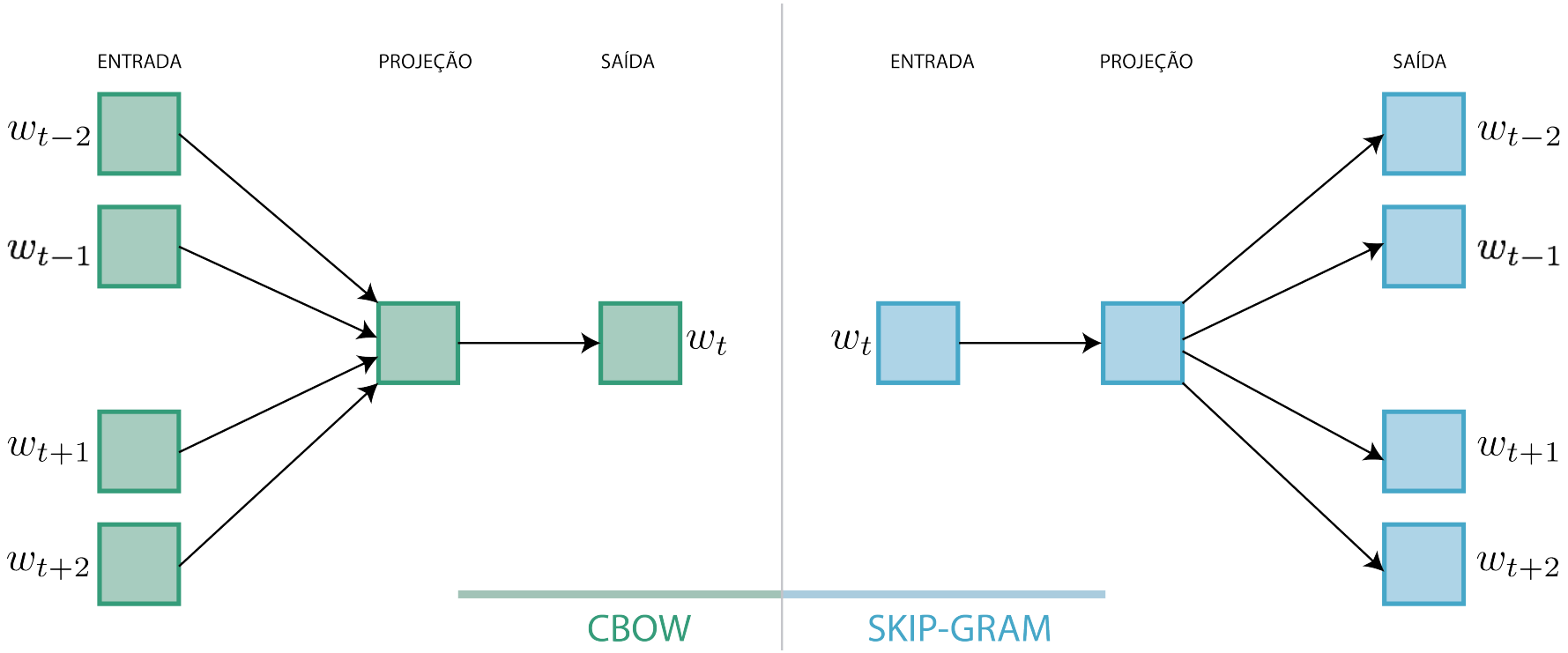
CBOW

CBOW (Contínuos Bag-of-Words): Permite fazer a predição de uma palavra qualquer do vocabulário, em uma posição específica, com base nas palavras presentes na sua vizinhança. Em outras palavras, determinar uma palavra com base no contexto vizinho. Diferente do Bag of Word convencional, ele usa valores contínuos para representar o contexto.

Skip-Gram

Skip-Gram: O objetivo deste método é encontrar representações de palavras que sejam úteis para predizer palavras vizinhas. Em outras palavras, usar uma palavra central para predizer as palavras que a circundam, com base em um intervalo fixo. O algoritmo dá mais importância para as palavras que estão mais próximas, considerando que as palavras mais afastadas estão menos relacionadas com a palavra selecionada. O Skip-Gram costuma se sair muito bem em tarefas onde é importante modelar regularidades semânticas. Esse algoritmo foi posteriormente estendido para considerar representações vetoriais de frases com o objetivo de modelar expressões idiomáticas, que não são meras composições de palavras.

Word2Vec


ENTRADA
PROJEÇÃO
SAÍDA
w_{t-2}
w_{t-1}
w_{t+1}
w_{t+2}
w_t
CBOW
ENTRADA
PROJEÇÃO
SAÍDA
w_t
w_{t-2}
w_{t-1}
w_{t+1}
w_{t+2}
SKIP-GRAM

FastText

FastText

Baseado no modelo Skip-Gram, o FastText parte da ideia de que os outros métodos relacionados ignoram a morfologia das palavras ao atribuir um vetor distinto para cada palavra. Trata-se de uma limitação porque, nestes casos, as representações para palavras raras deixam de ser obtidas, tornando o word embedding menos completo.

FastText

O FastText modela a morfologia das palavras, considerando sua sub-estrutura e modelando seus fragmentos como n-gramas. Assim, cada palavra é representada como uma conjunto de carácteres n-gramas, onde uma representação vetorial é atribuída a cada n-grama. Em seguida, as palavras são representadas pela soma das representações vetoriais desses n-gramas. O maior benefício do FastText é permitir obter representações vetoriais de palavras que eventualmente nem apareceriam no conjunto de aprendizagem.

FastText

$$armarinho =< ar, arma, mar, ar, inh, nho >$$