Inteligência Artificial para Robótica Móvel

Aprendizado de Máquina Profundo (Deep Learning)

Professor: Marcos Maximo

Roteiro

- Motivação;
- Evitando *overfitting* (regularização);
- *Dropout*;
- *Data Augmentation*;
- Normalização;
- *Vanishing/exploding gradients*;
- Melhorando inicialização dos pesos;
- Melhorias na Otimização;
- *Adam optimization*;
- *Batch normalization*;
- *Hyperparameter tunning*.
- Classificador *softmax*.

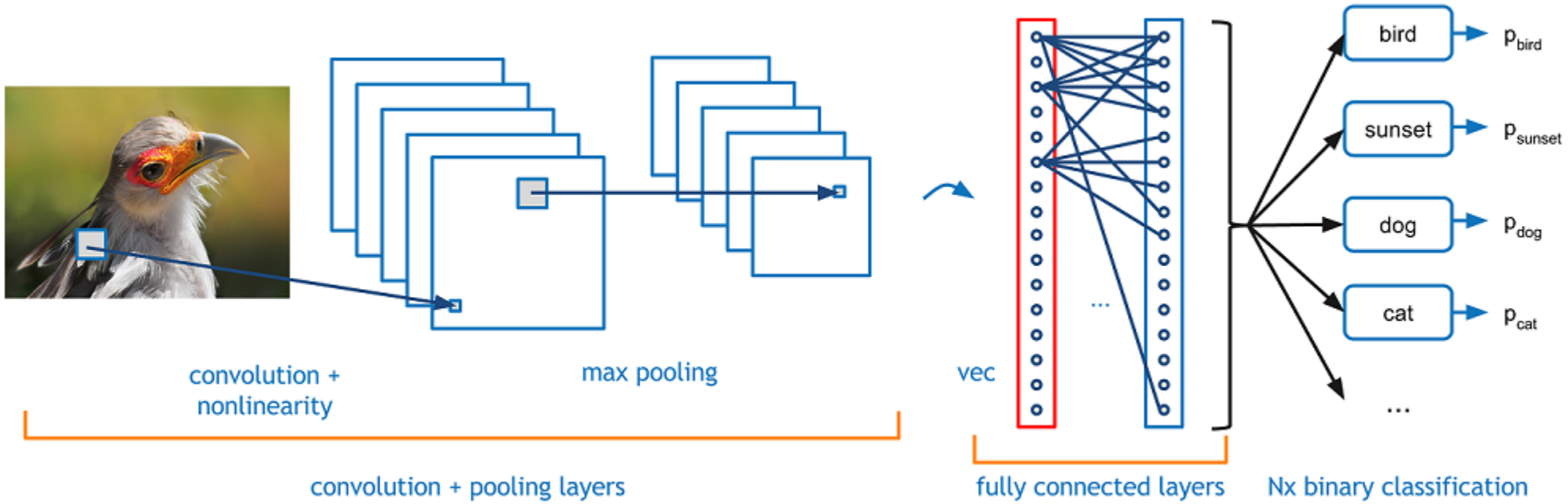
Motivação

Motivação

- Inglês: *Deep Learning* (DL).
- Foco da pesquisa em IA.
- Conjunto de técnicas que permitiram treinar redes neurais profundas.
- Muitas dessas técnicas são “heurísticas”.
- Algumas redes chegam a ter dezenas ou centenas de camadas.
- Com isso, algumas redes possuem milhões de parâmetros.
- Popularização de outras arquiteturas além da *feedforward fully-connected*.

Motivação

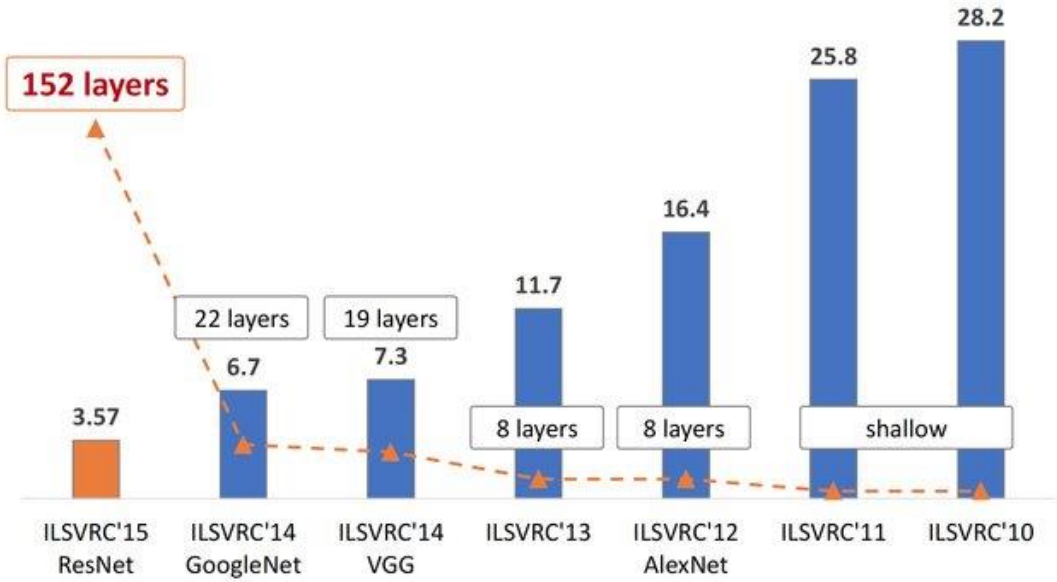
- Redes neurais convolucionais: uso de mascaras de convolução. Desempenho incrível em tarefas de visão computacional.



Fonte: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

Motivação

- Abordagem popularizada com a rede AlexNet (2012).
- Vencedores das últimas edições da competição ImageNet tem sido sempre CNNs, cada vez mais profundas.



Motivação

- Redes recorrentes: uso de realimentação (*loop*). Desempenho incrível para tarefas que exigem memória ou envolvem sequências.
- LSTM: Long Short Term Memory.
- Aplicações:
  - Reconhecimento de linguagem natural.
  - Predição de série temporal.
  - Sistema de decisão em que memória é importante (sistema de controle, tomada de decisão em jogos etc.).

Motivação

- Além das técnicas que veremos aqui, outros fatores tem sido importantes para crescimento de DL.
- Grande aumento na quantidade de dados disponível.
- Evolução de *hardware* paralelo (redes neurais são muito paralelizáveis): uso de GPU para treinar redes.
- Nova tendência é desenvolvimento de *hardware* específico para redes neurais.
- Grandes *frameworks* para redes neurais, desenvolvidas por especialistas.

Evitando *Overfitting* (Regularização)

Evitando *Overfitting*

- Redes profundas tem muito mais parâmetros que redes rasas.
- *Overfitting* torna-se um problema muito maior.
- É necessário muito mais dados para a rede generalizar, mas nem sempre isso é possível.
- Já vimos a ideia de usar regularização para penalizar pesos muito grandes, i.e. adiciona-se na função de custo:

Regularização L_2 (mais usada): $J_{L_2}(\boldsymbol{\theta}) = \frac{\lambda}{2m}\sum_j \theta_j^2$

Regularização L_1: $J_{L_1}(\boldsymbol{\theta}) = \frac{\lambda}{m}\sum_j |\theta_j|$

Regularização L_2

- Uma outra forma de interpretar regularização L_2:

$$\frac{\partial J}{\partial \theta_j} = \frac{\partial J_{error}}{\partial \theta_j} + \frac{\lambda}{m}\theta_j$$

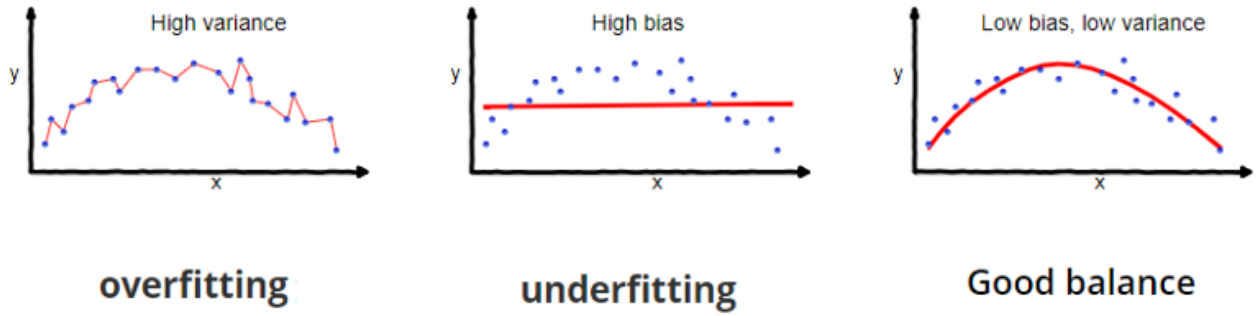
$$\theta_j = \theta_j - \alpha\frac{\partial J}{\partial \theta_j} = \theta_j - \alpha\frac{\lambda}{m}\theta_j - \alpha\frac{\partial J_{error}}{\partial \theta_j}$$

$$\theta_j = \left(1 - \alpha\frac{\lambda}{m}\right)\theta_j - \alpha\frac{\partial J_{error}}{\partial \theta_j}$$

"weight decay"

Bias-Variance Trade-off

- Conceito bem conhecido em ML.
- A partir de um certo ponto, bias (*underfitting*) e variância (*overfitting*) torna-se um *trade-off*.
- Intuitivamente, pode-se pensar que quando um modelo tenta *fittar* perfeitamente um dataset, ele acaba *fittando* também o ruído.



Dropout

Dropout

- Uma outra forma de regularização é *dropout regularization*.
- *Dropout*: ignorar (trocar por 0) a saída de um neurônio com certa probabilidade durante o treinamento.
- Isso “força” a rede a não depender demais em um único neurônio.
- Pode-se ter diferentes probabilidades para cada camada.

Dropout

- Forward propagation durante o treinamento:

$$\mathbf{a}^{[l]} = g^{[l]}\left(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}\right)$$

$$\mathbf{a}^{[l]} = \mathbf{a}^{[l]}.* \left(U\left(\left[\mathbf{0}_{n_l}, \mathbf{1}_{n_l}\right]\right) < p_{keep}^{[l]}\right)$$

em que $p_{keep}^{[l]}$ é a probabilidade de manter o neurônio no *dropout* (na camada l) e $.*$ é operação de multiplicação elemento a elemento.

- Forward propagation durante inferência:

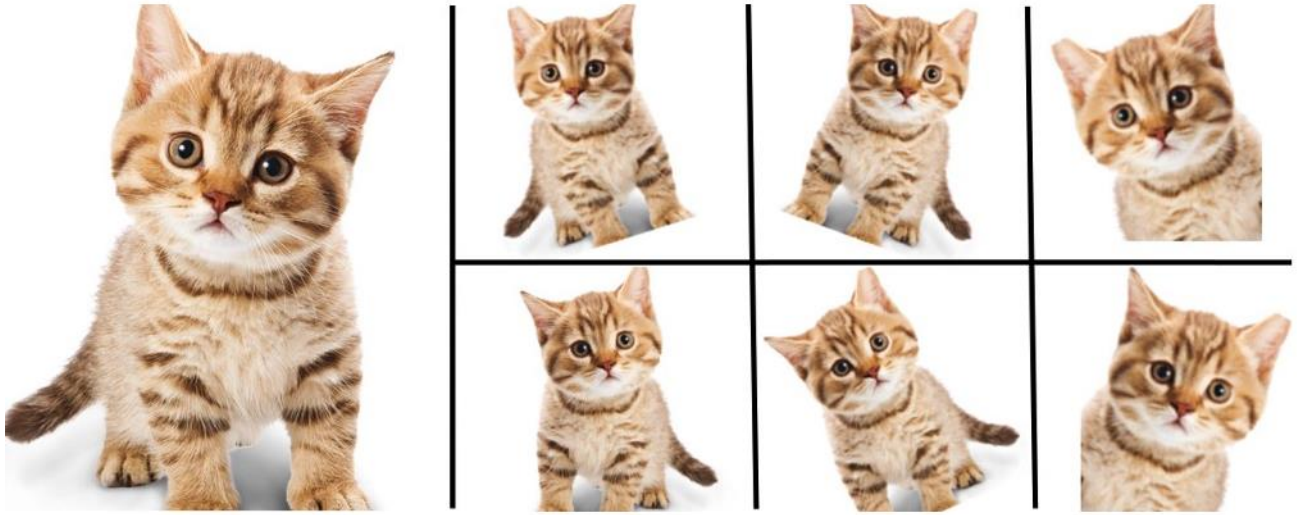
$$\mathbf{a}^{[l]} = \frac{1}{p_{keep}^{[l]}} g^{[l]}\left(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}\right)$$

- A correção é necessária pois o *dropout* reduz o valor esperado de $\mathbf{a}^{[l]}$.

Data Augmentation

Data Augmentation

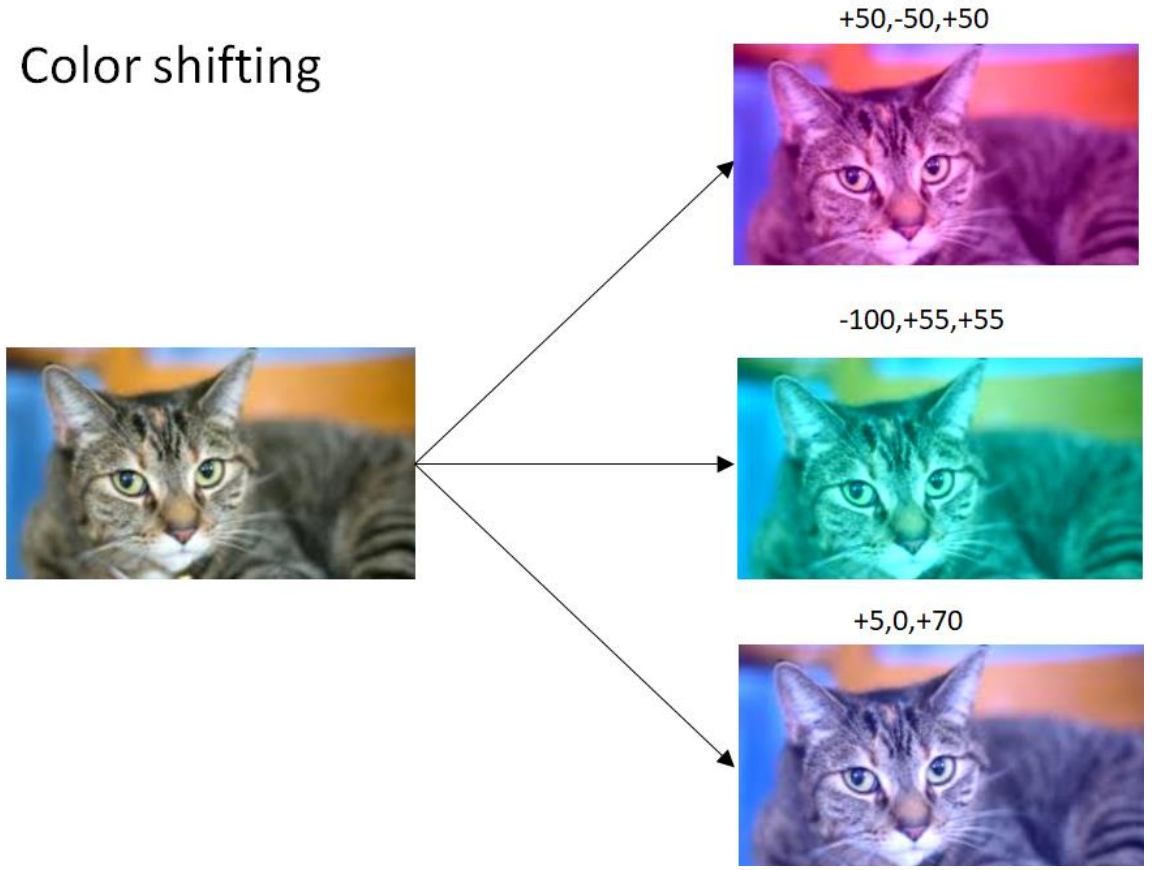
- Ideia: gerar dados artificialmente a partir do que você tem.



Fonte: https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced

- São todos gatinhos diferentes para uma rede neural.

Data Augmentation



Normalização

Normalização

- Às vezes as coordenadas dos seus dados tem valores muito diferentes entre si, de modo que fica ruim para a otimização.
- Além disso, algumas funções de ativação, como tangente hiperbólica ou sigmóide, saturam com valores muito grandes.
- ReLu muda de comportamento dependendo se é positivo ou negativo.
- Para evitar esses problemas, usa-se normalização dos dados de entrada.

Normalização

- Valores de entrada são trocados por (usando todo o *training set*):

$$x_j = \frac{x_j - \mu_j}{\sigma_j}$$

em que:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x_j^{(i)} - \mu_j\right)^2$$

Normalização

- **Observação:** é necessário guardar os valores de μ e σ usados no treinamento para usar depois durante a inferência.

Vanishing/Exploding Gradients

Vanishing/Exploding Gradients

- Imagine uma rede muito profunda, com várias camadas.
- Considere função de ativação linear em cada camada e despreze os biases.
- A ativação em uma camada l da rede é dada por:

$$\mathbf{a}^{[l]} = \mathbf{W}^{[l]}\mathbf{W}^{[l-1]}\mathbf{W}^{[l-2]} \dots \mathbf{W}^{[1]}\mathbf{x}$$

- Considere ainda que há apenas 1 neurônio por camada:

$$a^{[l]} = w^{[l]}w^{[l-1]}w^{[l-2]} \dots w^{[1]}x$$

- Se $w^{[l]} > 1, \forall l$, então a ativação da rede "explode", o que fará com os gradientes também explodam.
- Se $w^{[l]} < 1, \forall l$, então a ativação da rede "desaparece", o que fará com que os gradientes também desapareçam.

Melhorando Inicialização dos Pesos

Melhorando Inicialização dos Pesos

- Na aula anterior, foi sugerido:

$$w_{jk}^{[l]} \sim 0{,}001 * N(0,1)$$

$$b_j^{[l]} = 0$$

- Porém, isso não leva em conta o número de neurônios em uma camada. Lembre que:

$$z_j^{[l]} = \sum_{k=1}^{n^{[l-1]}} w_{jk}^{[l]}\, a_k^{[l-1]} + b_j^{[l-1]}$$

- Por conta disso, a variância de $z_j^{[l]}$ aumenta com o número de neurônios na camada $l-1$, i.e. $n^{[l-1]}$.

Melhorando Inicialização dos Pesos

- [Andrew Ng]: é possível melhorar o treinamento com uma inicialização que depende do número de neurônios da camada (*Xavier Initialization*):

$$w_{jk}^{[l]} \sim 0{,}001 N\left(0, \frac{1}{n^{[l-1]}}\right) = 0{,}001 \sqrt{\frac{1}{n^{[l-1]}}} N(0{,}1)$$

- Essa inicialização funciona bem para $tanh$. Para ReLU, usar:

$$w_{jk}^{[l]} \sim 0{,}001 \sqrt{\frac{2}{n^{[l-1]}}} N(0{,}1)$$

Melhorando Inicialização dos Pesos

- Um artigo mais moderno recomenda:

$$w_{jk}^{[l]} \sim 0{,}001 \sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}\, N(0,1)$$

- Essas maneiras de inicializar os pesos são conhecidas por diminuir o problema de *vanishing/exploding gradients* (Andrew Ng).

Melhorias na Otimização

Escolha do Tamanho do *Mini-Batch*

- Recomenda-se testar potências de 2 (Ng): 64, 128, 256, 512...
- Quanto maior, melhor se aproveita vetorização.
- Muitas vezes definido pelo o que cabe na memória da CPU ou GPU.

Agendamento da Taxa de Aprendizado

- É comum agendar o decrescimento da taxa de aprendizado α.
- Começa-se com valor alto e reduz-se a taxa à medida que a otimização prossegue.
- Pode-se definir uma equação:

$$\alpha_t = \frac{\alpha_0}{1 + \beta t}$$

$$\alpha_t = \alpha_0 e^{-\beta t}$$

- Também é comum definir "na mão": $\alpha = 0{,}01$ durante 10k épocas, depois $\alpha = 0{,}005$ durante mais 5k épocas, então $\alpha = 0{,}001$ durante 5k épocas finais.

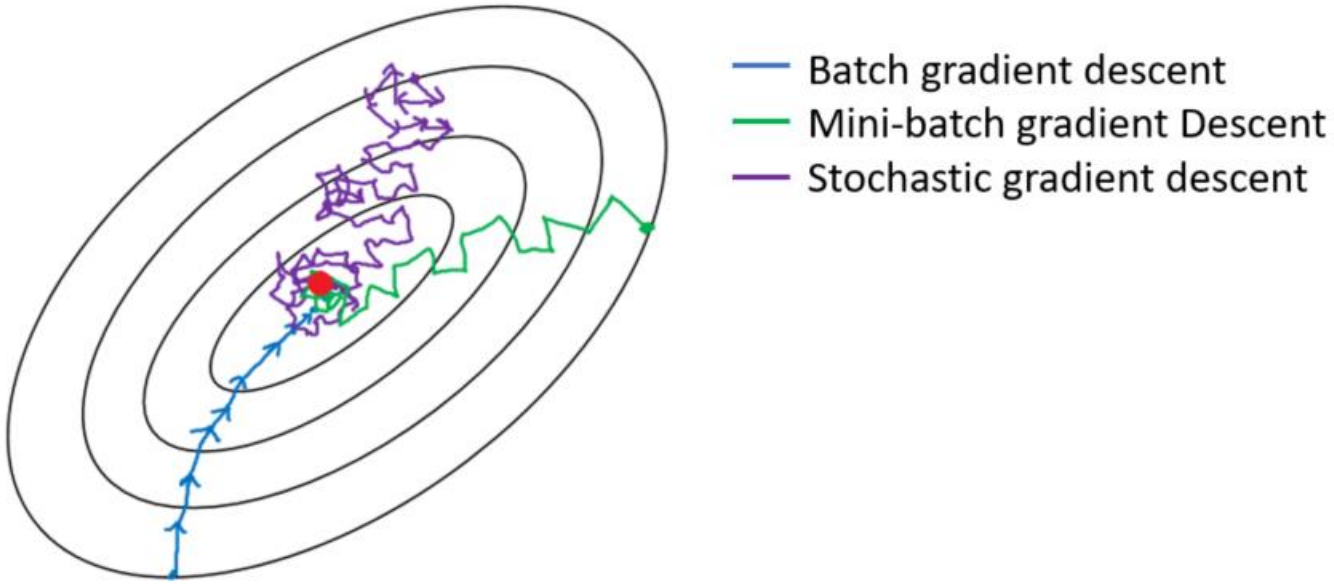
Problema de Ótimo Local

- Em otimização, estudamos o problema de ótimo local.
- Em redes neurais profundas, às vezes há milhões de parâmetros, logo pode-se esperar uma quantidade enorme de ótimos locais.
- Pesquisadores mostraram que na verdade quando se tem um espaço com tantos parâmetros, na maioria dos pontos em que o gradiente dá zero, tem-se um ponto de sela.
- Logo, o algoritmo não fica preso nesses pontos e não há tanto problema de ótimo local em redes neurais profundas quanto seria de se esperar.

Adam Optimization

Descida de Gradiente Estocástica

- Conforme visto na aula passada, o uso de *mini-batch* torna o gradiente ruidoso.



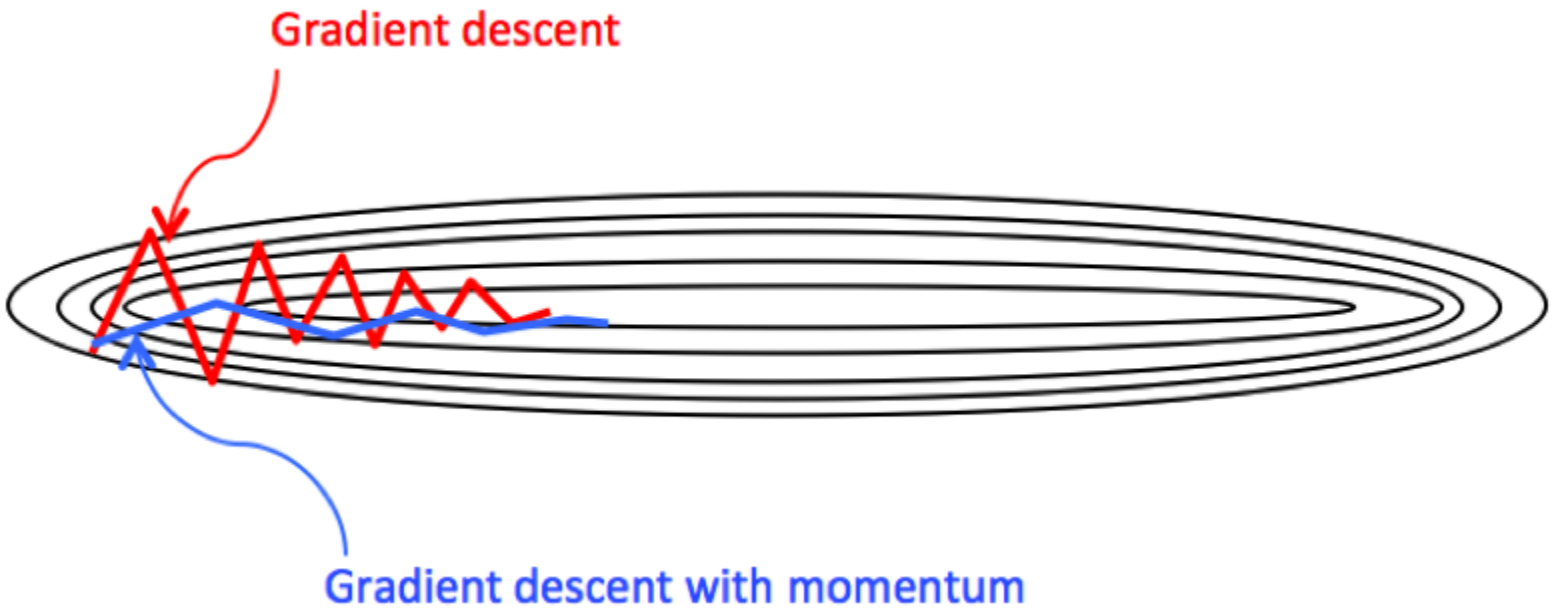
Momento

- Gradiente é ruidoso.
- O truque que aprendemos para reduzir ruído é filtrar.
- Usar média exponencial no cálculo do gradiente:

$$\mathbf{v}_{dW} = \beta \mathbf{v}_{dW} + (1 - \beta)\mathbf{dW}$$
$$\mathbf{v}_{db} = \beta \mathbf{v}_{db} + (1 - \beta)\mathbf{db}$$
$$\mathbf{W} = \mathbf{W} - \alpha \mathbf{v}_{dW}, \mathbf{b} = \mathbf{b} - \alpha \mathbf{v}_{db}$$

- $\mathbf{dW}$ é o gradiente dos pesos e $\mathbf{db}$ é o gradiente dos biases.
- β é mais outro hiperparâmetro.

Momento



Fonte: https://medium.com/machine-learning-bites/deeplearning-series-deep-neural-networks-tuning-and-optimization-39250ff7786d

Correção de Bias

- Observação: média exponencial possui bias no começo, se inicializar com $y_0 = 0$.
- É possível corrigir esse bias fazendo:

$$y_t = \beta y_{t-1} + (1 - \beta) x_t$$

$$y'_t = \frac{y_t}{1 - \beta^t}$$

Correção de Bias

- Inicializando $y_0 = 0$:

$$y_1 = \beta y_0 + (1-\beta)x_1 = (1-\beta)x_1$$

$$y_1' = \frac{(1-\beta)x_1}{1-\beta} = x_1$$

$$y_2 = (1-\beta)\beta x_1 + (1-\beta)x_2$$

$$y_2' = \frac{y_2}{1-\beta^2} = \frac{(1-\beta)\beta x_1 + (1-\beta)x_2}{1-\beta^2}$$

- Perceba que y_2' é média ponderada de x_1 e x_2 com pesos $(1-\beta)\beta$ e $(1-\beta)$.
- Na prática, é comum não se fazer correção de bias, porque esse bias desaparece rápido.

RMSProp

- RMS = Root Mean Square.
- Baseia-se no cálculo de um “RMS”:

$$\mathbf{s}_{dW} = \beta_2 \mathbf{s}_{dW} + (1 - \beta_2)(\mathbf{dW} ** 2)$$
$$\mathbf{s}_{db} = \beta_2 \mathbf{s}_{db} + (1 - \beta_2)(\mathbf{db} ** 2)$$
$$\mathbf{W} = \mathbf{W} - \alpha \frac{\mathbf{dW}}{\sqrt{\mathbf{s}_{dW}} + \varepsilon}, \mathbf{b} = \mathbf{b} - \alpha \frac{\mathbf{db}}{\sqrt{\mathbf{s}_{db}} + \varepsilon}$$

- $\mathbf{a} ** 2$ representa elevar ao quadrado elemento a elemento. Demais operações também são elemento a elemento.
- ε evita divisão por zero; valor padrão é $\varepsilon = 10^{-8}$.
- β_2 é um hiperparâmetro.

Adam Optimization

- Adam: *Adaptive Moment Estimation*.
- Autor: Adam Coates.
- Junta momento e RMSProp.
- Um dos melhores algoritmos de descida de gradiente na prática.
- Muito popular em *Deep Learning*.

Adam Optimization

- $\mathbf{v}_{dW} = \mathbf{0}, \mathbf{s}_{dW} = \mathbf{0}, \mathbf{v}_{db} = \mathbf{0}, \mathbf{s}_{db} = \mathbf{0}$.
- Na iteração t:

$$\mathbf{dW}, \mathbf{db} = backpropagation(NN)$$

$$\mathbf{v}_{dW} = \beta_1 \mathbf{v}_{dW} + (1 - \beta_1)\mathbf{dW}, \mathbf{v}_{db} = \beta_1 \mathbf{v}_{db} + (1 - \beta_1)\mathbf{db}$$

$$\mathbf{s}_{dW} = \beta_2 \mathbf{s}_{dW} + (1 - \beta_2)\mathbf{dW} ** 2, \mathbf{s}_{db} = \beta_2 \mathbf{s}_{db} + (1 - \beta_2)\mathbf{db} ** 2$$

$$\mathbf{v}'_{dW} = \frac{\mathbf{v}_{dW}}{1 - \beta^t}, \mathbf{s}'_{dW} = \frac{\mathbf{s}_{dW}}{1 - \beta^t}, \mathbf{v}'_{db} = \frac{\mathbf{v}_{db}}{1 - \beta^t}, \mathbf{s}'_{db} = \frac{\mathbf{s}_{db}}{1 - \beta^t}$$

$$\mathbf{W} = \mathbf{W} - \alpha \frac{\mathbf{v}'_{dW}}{\sqrt{\mathbf{s}'_{dW}} + \varepsilon}, \mathbf{b} = \mathbf{b} - \alpha \frac{\mathbf{v}'_{db}}{\sqrt{\mathbf{s}'_{db}} + \varepsilon}$$

- **Observação:** operações elemento a elemento.

Adam Optimization

- Um motivo para Adam ser tão popular é que os valores padrões dos hiperparâmetros já funcionam muito bem.

$$\beta_1 = 0{,}9$$
$$\beta_2 = 0{,}999$$
$$\varepsilon = 10^{-8}$$

α ainda depende do problema

Batch Normalization

Batch Normalization (BN)

- Anteriormente, argumentamos que normalizar os dados de entrada ajuda no treinamento da rede.
- Porém, o mesmo problema de antes ainda pode existir nas camadas intermediárias.
- A ativação de uma camada é entrada da próxima.
- Pode-se fazer normalização de $\mathbf{z}^{[l]}$ ou $\mathbf{a}^{[l]}$:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{a}^{[l]} = g(\mathbf{z}^{[l]})$$

- O mais comum é fazer normalização de $\mathbf{z}^{[l]}$ (Andrew Ng).

Batch Normalization (BN)

- Para cada mini-batch B:

$$\mu_j = \frac{1}{m_B}\sum_{i=1}^{m_B} z_j^{[l](i)}, \qquad \sigma_j^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(z_j^{[l](i)} - \mu\right)^2$$

$$z_{norm,j}^{[l]} = \frac{z_j^{[l](i)} - \mu_B}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$\tilde{z}_j^{[l]} = \gamma_j z_{norm,j}^{[l]} + \beta_j$$

- γ_j e β_j define nova média e covariância (parâmetros a serem aprendidos).
- Pode-se pensar em BN como uma nova camada.

Inferência com *Batch Normalization*

- Na inferência, é necessário levar em conta que treinamento aconteceu com BN.
- Andrew Ng: guardar média móvel de $\boldsymbol{\mu}^{[l]}$ e $\boldsymbol{\sigma}^{[l]}$ dos mini-batches e usar esses valores para normalizar na inferência.
- Outra maneira: usar as estatísticas de todo o *training set*.

Hyperparameter Tunning

Hiperparâmetros

- Taxa de aprendizado α.
- Hiperparâmetros do Adam β_1, β_2, ε: em geral não precisa mexer.
- Número de camadas.
- Número de neurônios em cada camada.
- Agendamento da taxa de aprendizado.
- Tamanho do *mini-batch*.

Tunando Hiperparâmetros

- Testar todos os valores em um *grid* (ruim, pois demora demais).
- Definir intervalo para cada hiperparâmetro e amostrar aleatoriamente (mais usado).
- Normalmente, usa-se distribuição uniforme para amostragem.
- Para alguns parâmetros é melhor usar escala logarítmica, e.g. taxa de aprendizado.
- Após uma exploração inicial, pode-se refinar a busca em torno de uma região mais promissora.

Tunando Hiperparâmetros

- Pode-se pensar também em usar otimização, porém isso envolve custo computacional proibitivo em geral.
- Na prática, processo ainda depende muito de experiência e tentativa e erro.

Classificador Softmax

Classificador Softmax

- Na aula passada, apresentamos uma estratégia para classificação multi-classe codificando classes da seguinte forma:

Gato: $[1\ 0\ 0\ 0]^T$.

Cachorro: $[0\ 1\ 0\ 0]^T$.

Papagaio: $[0\ 0\ 1\ 0]^T$.

Tartaruga: $[0\ 0\ 0\ 1]^T$.

- Então, usava-se sigmoide em cada um dos neurônios de saída.

Classificador Softmax

- Atualmente, o mais comum é usar um tipo de camada conhecida por *softmax* na camada de saída:

$$a_j^{[L]} = \frac{e^{z_j^{[L]}}}{\sum_{k=1}^{n^{[L]}} e^{z_k^{[L]}}}$$

- A ideia é implementar uma função de max "suave".
- Perceba que a ativação de um neurônio depende de todos os neurônios da camada.

Classificador Softmax

- *Loss Function*:

$$\mathcal{L}\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right) = -\sum_{j=1}^{K} y_j^{(i)} \log\left(\hat{y}_j^{(i)}\right)$$

Para Saber Mais

- Especialização de *Deep Learning* do Andrew Ng no Cousera (curso de *Deep Learning*).
- Capítulos 6 a 8 do livro: GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. The MIT Press, 2016.

Laboratório 8

Laboratório 8

- Aprender Tensorflow.
- Implementar redes neurais com Tensorflow.
- Problema: *imitation learning* de movimento de robô humanoide.