Machine Learning Engineer Nanodegree

Uma comparação de modelos de *deep learning* para classificação de imagens

Paulo Viadanna - Junho de 2017

Índice

I. Definição

Visão geral do projeto

Com a rápida expansão da capacidade computacional da última década, o uso de modelos de aprendizado de máquina baseados em redes neurais vem revolucionando a computação tornando factíveis capacidades que antes estavam exclusivamente no campo da ficção científica.

Na vanguarda desse movimento está a tarefa de classificação de imagens, onde os modelos são capazes de identificar objetos, lugares, expressão facial e mais recentemente, correlação entre objetos de uma cena.

A idéia principal desse projeto é explorar modelos de aprendizado profundo, *deep learning*, para fazer a classificação de objetos em uma imagem.

Declaração do Problema

Como minha iniciação no campo de *deep learning* decidi explorar a tarefa de classificação de imagens. Sendo essa uma área muito bem explorada e utilizada nesse campo, não espero abrir novos caminhos em reconhecimento de imagens, mas sim avaliar as técnicas utilizadas para a construção de modelos e o efeito do tamanho dos mesmos.

Para isso utilizarei o banco de imagens CIFAR-10[1] que contém 60.000 imagens catalogadas distribuídas em 10 categorias. Utilizarei esse banco de imagens para treinar e avaliar a performance de diversos modelos utilizando diferentes técnicas na tentativa de melhorar a acurácia da classificação.

Métricas

A métrica mais comum para avaliar modelos de classificação é simplesmente a acurácia, basicamente se o modelo está certo ou errado em sua previsão, e essa será a métrica principal a ser utilizada aqui.

Outra métrica muito utilizada para a avaliação de classificação de imagens é a *Top-5 accuracy* que indica se a classe correta se encontra entre as cinco classificações de maior probabilidade feitas pelo modelo. Como o dataset utilizado contém imagens de apenas 10 classes diferentes, esta métrica poderia dar uma falsa impressão de alto desempenho, já que mesmo um modelo aleatório seria capaz de alcançar 50% de acurácia utilizando esta métrica.

---

[1] "CIFAR-10 and CIFAR-100 datasets." https://www.cs.toronto.edu/~kriz/cifar.html. Acessado em 8 jun. 2017.

II. Análise do Problema

Exploração do Conjunto de Dados

De acordo com o site de origem do conjunto de dados CIFAR-10[2], ele é composto de 60.000 imagens coloridas com tamanho 32x32 pixels.

Essas imagens estão divididas em dez classes mutuamente exclusivas: avião, automóvel, pássaro, gato, cervo, cão, sapo, cavalo, navio e caminhão.

A versão do conjunto de dados disponível para Python já contém a separação entre 50.000 imagens para treinamento e 10.000 para testes. Como vários modelos serão desenvolvidos, é necessário um terceiro conjunto de dados para validação, que foi criado selecionando-se 10.000 imagens do conjunto de dados de treinamento.

No notebook Jupyter anexo é feita uma verificação dessas informações para garantir que o conjunto de dados utilizado realmente tem esses atributos, visto que foi baixado através da biblioteca Keras[3].

Exploração Visual

Como conjunto de dados é composto de imagens, a primeira e mais óbvia visualização é das próprias imagens, para se ter uma idéia do que os modelos irão encontrar.

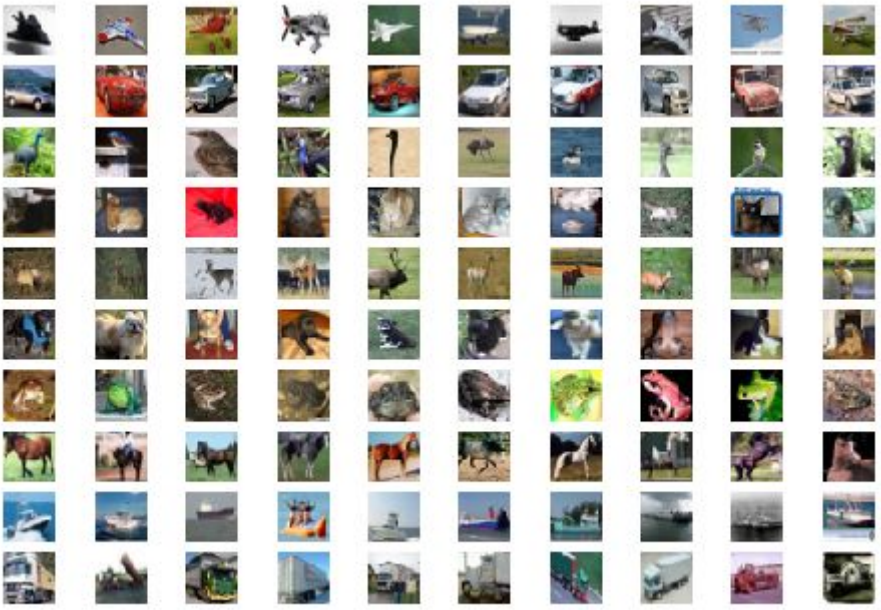


Figura 1. Dez exemplos de imagens de cada uma das dez classes

[2] "CIFAR-10 and CIFAR-100 datasets." https://www.cs.toronto.edu/~kriz/cifar.html. Acessado em 8 jun. 2017.
[3] "Datasets - Keras Documentation." https://keras.io/datasets/. Acessado em 8 jun. 2017.

As imagens não foram reduzidas para melhor apresentação, mas sim para aproximar o tamanho real delas. É possível perceber que se tratam de fotos dos objetos de cada classe.

Uma coisa interessante que se pode ver aqui é que no caso da classe aviões, tanto aviões reais quanto de brinquedos são incluídos. Outra coisa que me chamou a atenção é a variação encontrada em algumas classes, como a variedade de modelos de carros e navios presentes, a variação de cores das diversas raças de cavalos e de espécies e tamanhos de pássaros.

Em seguida, uma visualização da distribuição das classes, para se ter uma idéia se o conjunto de dados é balanceado ou não. Isso é extremamente importante pois um conjunto de dados de treinamento não balanceado irá introduzir um viés[4] no modelo gerado.

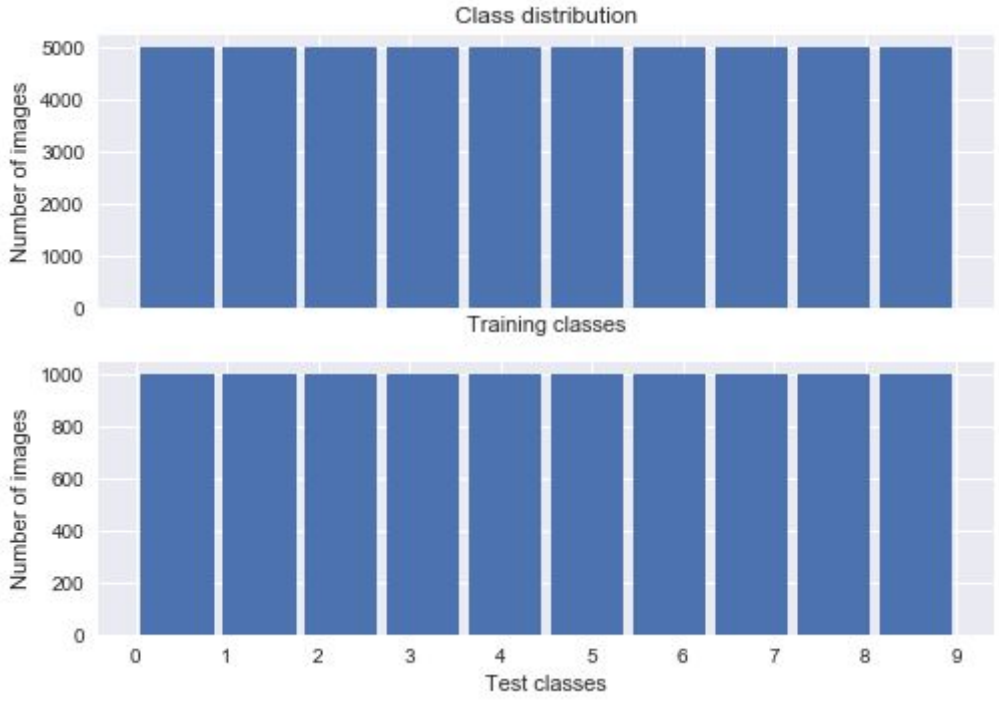


Figura 2. Distribuição das classes

Aqui, é possível perceber que ambos os conjuntos de dados são perfeitamente balanceados, o que tira a importância da discussão do viés em relação a determinadas classes.

[4] "The Impact of Imbalanced Training Data for Convolutional ... - KTH." 8 mai. 2015, https://www.kth.se/social/files/588617ebf2765401cfcc478c/PHensmanDMasko_dkand15.pdf. Acessado em 8 jun. 2017.

Modelos e Técnicas

Para avaliar a capacidade de aprendizado, elaborei um conjunto de modelos, indo desde um simples modelo com duas camadas até o enorme modelo Inception v3[5] que foi elaborado por engenheiros do Google em 2015[6] para a competição ImageNet[7]. A ideia é incrementalmente aumentar a capacidade de aprendizado e verificar o efeito na métrica a ser utilizada.

Foi utilizada a técnica de normalização[8] dos dados como pré-processamento, a fim de evitar que os pesos do modelo explodam ao serem continuamente multiplicados, além de também tornar esse um problema bem comportado, tornando mais fácil a tarefa do otimizador de encontrar uma solução.

Outra etapa de pré-processamento será a geração de novas imagens[9] aplicando rotação e *shift* para aumentar o tamanho do dataset de treinamento, criando imagens levemente diferentes das originais.

Por fim, a técnica de dropout[10] será avaliada em certos modelos para avaliar o ganho de performance com a redução da variância.

Benchmark

Como já dito anteriormente, o intuito deste projeto não é abrir novos caminhos em reconhecimento de imagens, mas sim fazer uma comparação de performance entre diversas possibilidade de modelos.

Sendo assim, a comparação a ser feita será realmente entre os modelos presente no projeto, e considerando que já existem[11] modelos rasos com 75% de acurácia, espero ao menos esse desempenho de algum dos modelos testados.

---

[5] "[1512.00567] Rethinking the Inception Architecture for Computer Vision." 2 dez. 2015, https://arxiv.org/abs/1512.00567. Acessado em 8 jun. 2017.
[6] "Research Blog: Train your own image classifier with Inception in ...." 9 mar. 2016, https://research.googleblog.com/2016/03/train-your-own-image-classifier-with.html. Acessado em 8 jun. 2017.
[7] "ImageNet." http://www.image-net.org/. Acessado em 8 jun. 2017.
[8] "Data Preprocessing - Ufldl - Deep Learning." http://ufldl.stanford.edu/wiki/index.php/Data_Preprocessing. Acessado em 8 jun. 2017.
[9] "Understanding data augmentation for classification ... - at www.arxiv.org.." https://arxiv.org/pdf/1609.08764. Acessado em 8 jun. 2017.
[10] "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." http://www.jmlr.org/papers/volume15/srivastava14a.old/source/srivastava14a.pdf. Acessado em 8 jun. 2017.
[11] "Classification datasets results - Rodrigo Benenson." http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html. Acessado em 8 jun. 2017.

III. Metodologia

Pré-processamento dos Dados

Conforme mencionado, as matrizes contendo as imagens foram normalizadas para facilitar a convergência do otimizador e evitar a explosão dos pesos.

Essa normalização consiste em centralizar os valores da matriz calculando-se a média de todos os valores e subtraindo esse valor de cada item. Em seguida cada elemento foi dividido pelo valor máximo subtraído do valor mínimo para que os valores finais de cada elemento fique entre [-1, 1].

O vetor contendo as classes também foi pré-processado através de One-Hot encoding[12] que converte cada inteiro em uma matriz de binários[13].

Como o dataset original é composto de treinamento e testes, uma parte do conjunto de treinamento é separada para a etapa de validação dos modelos. São separadas 10.000 imagens escolhidas aleatoriamente para validação, mantendo-se a mesma distribuição de classes do conjunto de treinamento original usando-se uma técnica chamada estratificação[14].

Por fim, o conjunto de treinamento terá 40.000 imagens mas usando o ImageDataGenerator[15] da biblioteca Keras é possível aumentar esse número para um valor arbitrário. Além disso, as novas imagens geradas podem ser ligeiramente diferentes das originais ao invés de uma simples cópia. Assim, cada nova imagem gerada poderá ser rotacionada até 10º em ambas as direções, ter os pixels movidos em até 10% em relação ao eixos x e y e ter um zoom de até 10% aplicado.

Todas essas técnicas estão implementadas no arquivo *preprocessing.py*[16] e para agilizar o desenvolvimento e testes dos modelos, as imagens são pré-processadas (e aumentadas se necessário) apenas na primeira vez em que certo tamanho de conjunto de dados de treinamento é requerido. Assim que geradas, as matrizes são salvas e reutilizadas em chamadas posteriores.

---

[12] "One-Hot Encoding." http://www.cs.toronto.edu/~guerzhoy/321/lec/W04/onehot.pdf. Acessado em 9 jun. 2017.
[13] "One-Hot Encoding - YouTube." 6 jun. 2016, https://www.youtube.com/watch?v=2Uyr93f3C2M. Acessado em 9 jun. 2017.
[14] "Stratified sampling - Wikipedia." https://en.wikipedia.org/wiki/Stratified_sampling. Acessado em 9 jun. 2017.
[15] "Image Preprocessing - Keras Documentation." https://keras.io/preprocessing/image/. Acessado em 9 jun. 2017.
[16] “preprocessing.py in Github” https://github.com/viadanna/cifar/blob/master/preprocessing.py Acessado em 9 jun. 2017.

Implementação e Refino

Como a idéia desse projeto é testar diversos modelos, iniciei o desenvolvimento com o arquivo *experiment.py*[17] que basicamente define as funcionalidades comuns para todos os modelos, incluíndo receber argumentos para o teste, carregar os conjuntos de dados, efetuar o treinamento do model, salvar e carregar o modelo treinado e por fim avaliar o conjunto de testes.

Em seguida comecei o desenvolvimento dos modelos de *deep learning* propriamente ditos. Para isso criei um arquivo *models.py*[18] contendo funções que retornam um modelo para poder executar facilmente todos os modelos.

Modelo Simple

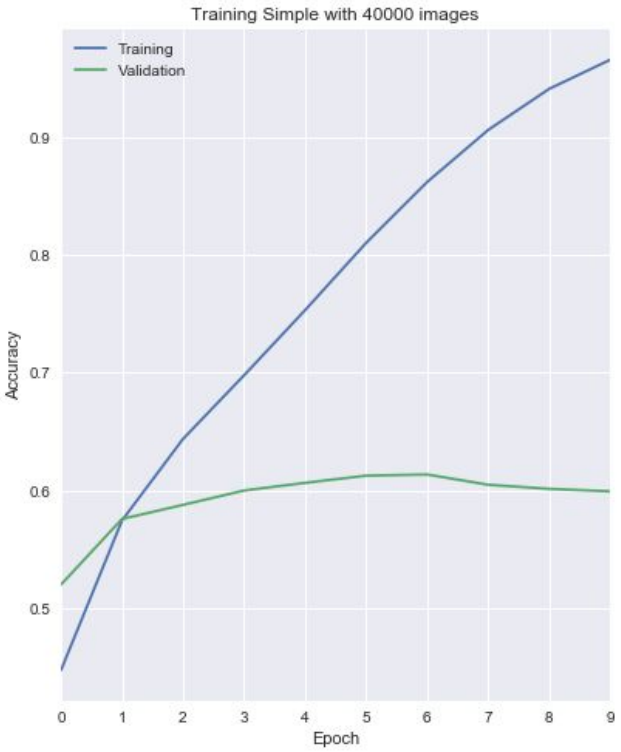
Comecei com a CNN mais simples que consegui, que contém uma camada convolucional e outra densamente conectada. Abaixo segue uma tabela com as camadas, seu parâmetros e o tamanho da saída de cada uma delas.

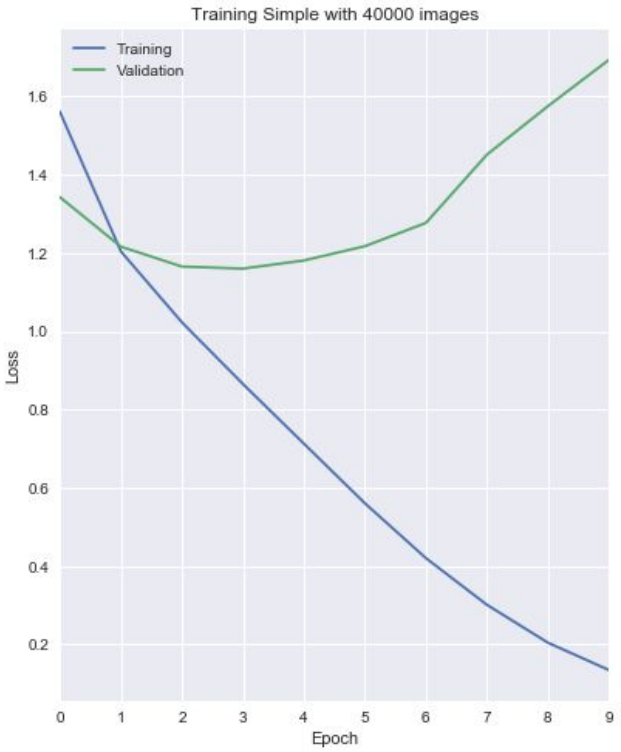
Tipo de camada	Parâmetros	Entrada	Saída
Convolucional	Kernel 3x3	32x32x3	30x30x16
Flatten		30x30x16	14400
Fully Connected		14400	512
Saída		512	10

Tabela 1 - Modelo Simple

Esse modelo foi então treinado com o conjunto de dados original de 40.000 imagens e rapidamente apresentou alta variância, como pode ser visto no gráfico abaixo. Criei então uma versão alternativa adicionando uma camada de Max Pooling[19] e outra de Dropout na tentativa de reduzir a variância.

[17] “experiment.py in Github” https://github.com/viadanna/cifar/blob/master/experiment.py Acessado em 9 jun. 2017
[18] “models.py in Github” https://github.com/viadanna/cifar/blob/master/models.py Acessado em 9 jun. 2017
[19] "Fully-Connected Layer - CS231n Convolutional Neural Networks for ...." http://cs231n.github.io/convolutional-networks/. Acessado em 12 jun. 2017.





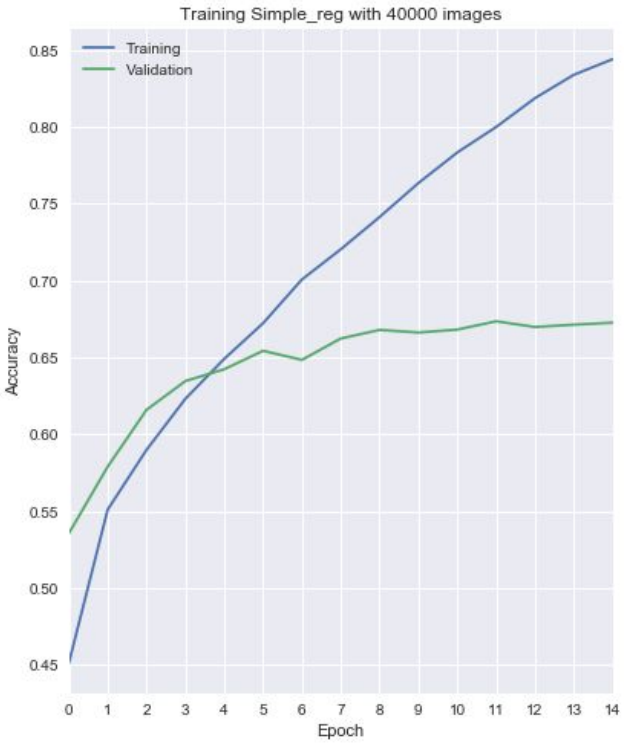
Modelo Simple com regularização (simple_reg)

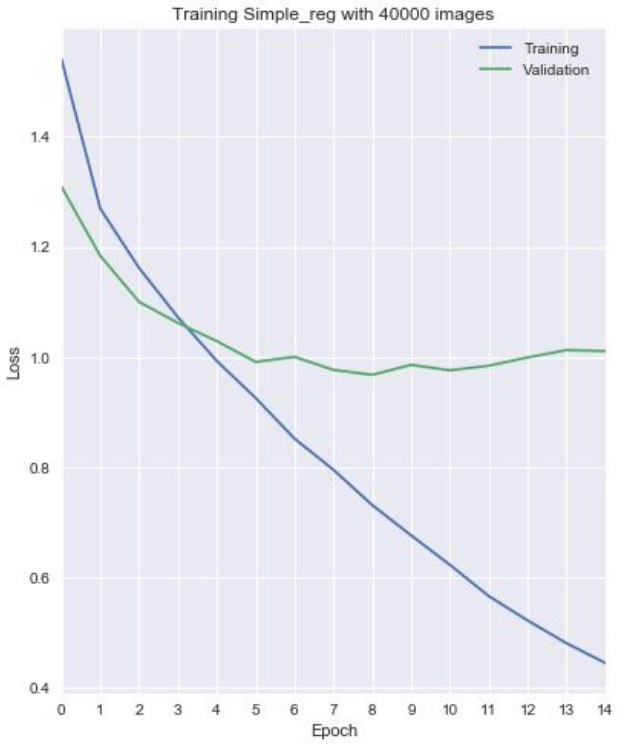
Como mencionado acima, o modelo rapidamente apresentou alta variância e decidi experimentar adicionar Max Pooling e Dropout para examinar o efeito dessas.

Tipo de camada	Parâmetros	Entrada	Saída
Convolucional	Kernel 3x3	32x32x3	30x30x16
Max Pooling	Stride 2x2	30x30x16	15x15x16
Flatten		15x15x16	3600
Dropout	Keep 0.5	3600	3600
Fully Connected		3600	512
Saída		512	10

Tabela 2 - Modelo Simple com regularização

Houve uma pequena melhora no resultado de validação, como pode ser visto no gráfico de treinamento, mas ainda muito aquém do esperado. Obviamente é hora de experimentar modelos mais profundos.





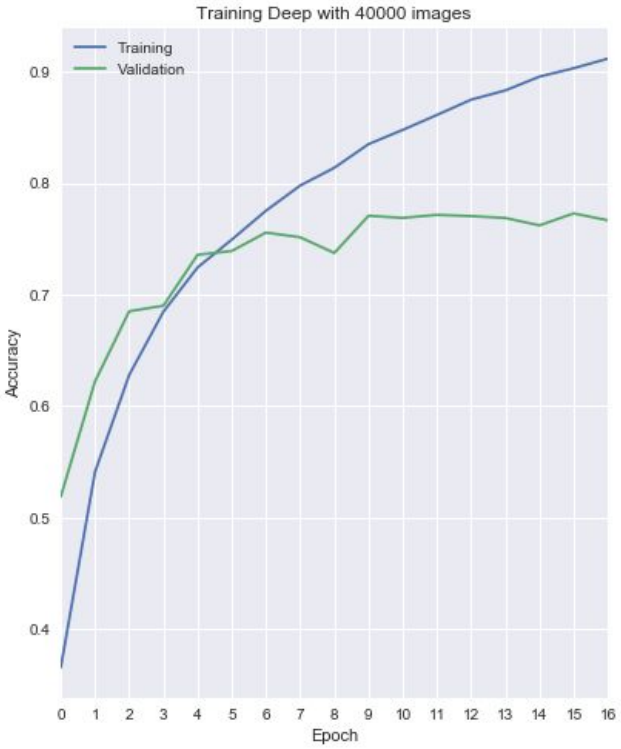
Modelo Deep

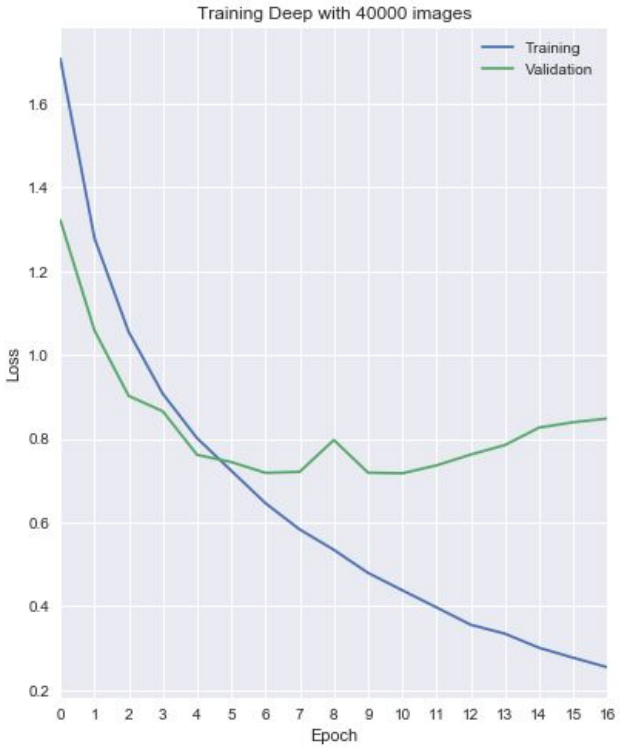
Em seguida, criei esse modelo com um maior número de camadas e utilizando as mesmas técnicas para evitar o *overfitting*.

Tipo de camada	Parâmetros	Entrada	Saída
Convolucional	Kernel 3x3	32x32x3	30x30x64
Convolucional	Kernel 3x3	30x30x64	28x28x64
Max Pooling	Stride 2x2	28x28x64	14x14x64
Convolucional	Kernel 3x3	14x14x64	12x12x128
Convolucional	Kernel 3x3	12x12x128	10x10x128
Max Pooling	Stride 2x2	10x10x128	5x5x128
Flatten		5x5x128	3200
Fully Connected		3200	256
Fully Connected		256	256
Saída		256	10

Tabela 3 - Modelo Deep

Esse modelo já apresentou uma melhora significativa durante o processo de treinamento.





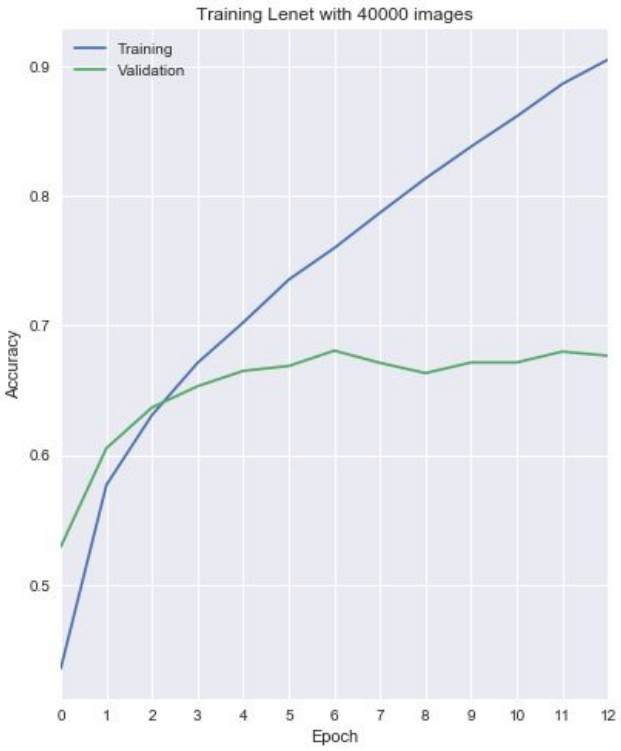
Modelo LeNet-5

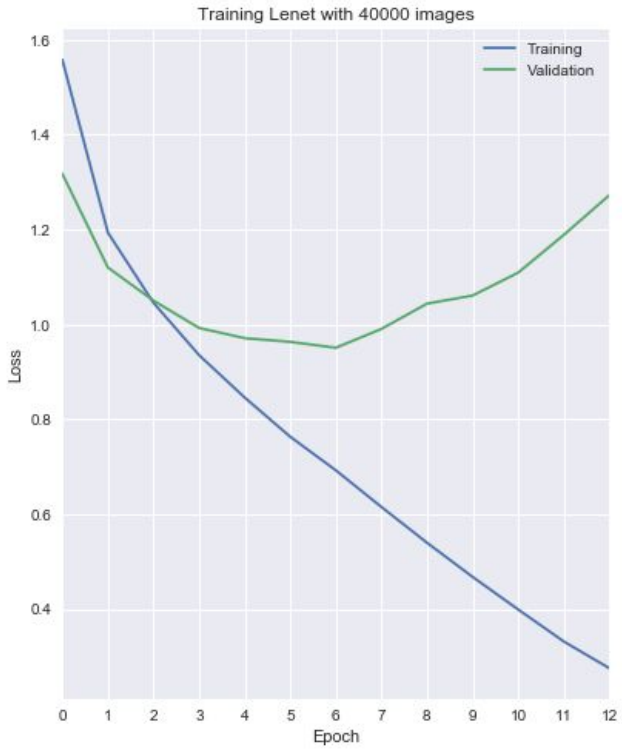
Durante minhas pesquisas me deparei com a arquitetura **LeNet-5**[20] que achei ser bem interessante e também a implementei para comparar os resultados dos meus modelos com alguma arquitetura já bem conhecida. Neste caso também percebi alta variância e uma versão do modelo também utilizando Dropout para reduzi-la foi testada.

Tipo de camada	Parâmetros	Entrada	Saída
Convolucional	Kernel 3x3	32x32x3	30x30x16
Max Pooling	Stride 2x2	30x30x16	15x15x16
Convolucional	Kernel 3x3	15x15x16	13x13x32
Max Pooling	Stride 2x2	13x13x32	6x6x32
Flatten		6x6x32	1152
Fully Connected		1152	256
Fully Connected		256	128
Saída		128	10

Tabela 4 - Modelo LeNet-5

[20] "LeNet-5 - Yann LeCun." http://yann.lecun.com/exdb/lenet/. Acessado em 12 jun. 2017.





Pode-se perceber que essa arquitetura não foi capaz de alcançar a mesma performance da anterior, mas considerando que essa é uma arquitetura rasa, achei o resultado interessante.

Hiperparâmetros

A partir daí comecei a enfrentar uma dificuldade não esperada: modelos maiores não cabiam na memória da GPU que eu estava utilizando para executar os modelos. Em alguns casos a própria arquitetura não cabia na memória, não importando então o *batch size*.

Este é um bom momento para discutir os parâmetros de treinamento dos modelos. Em minhas primeiras tentativas, utilizei um número fixo de épocas, mas logo percebi que isso não funcionaria para treinar diferentes modelos.

Passei então a utilizar a técnica de *early stopping*, onde ao final de cada época de treinamento, é considerado se o *validation loss* teve alguma melhoria. Depois de cinco épocas sem melhoria o treinamento é interrompido. Esta técnica também é bem interessante para evitar o treinamento excessivo que pode resultar em alta variância.

Com relação ao batch size, ou o número de imagens que são processadas em paralelo a cada passo de treinamento, empiricamente escolhi 128, pois oferece uma boa redução no tempo de treinamento, sem problemas de falta de memória na grande maioria das arquiteturas e sem perda significativa de acurácia[21].

[21] "Model Accuracy and Runtime Tradeoff in Distributed Deep Learning." 14 set. 2015, https://arxiv.org/abs/1509.04210. Acessado em 12 jun. 2017.

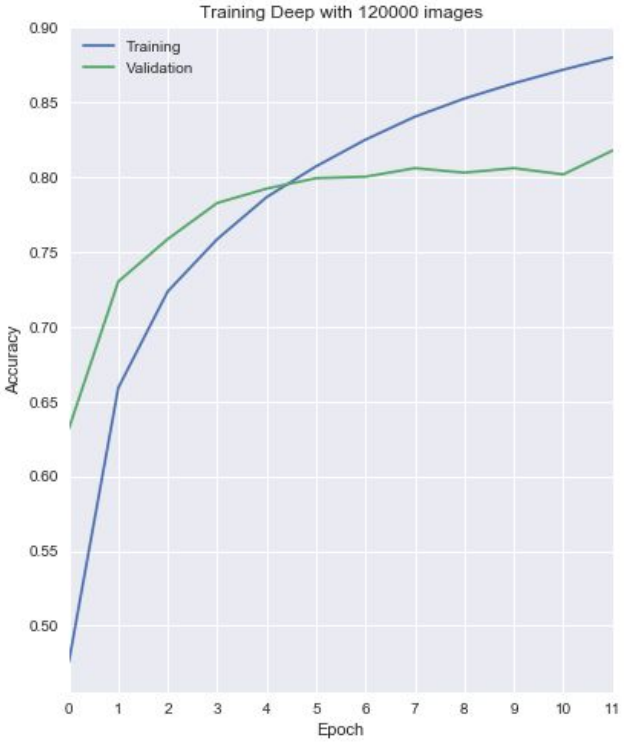
Outra medida para conseguir os melhores modelos foi salvar os pesos dos modelos sempre que houvesse uma melhora na métrica de validação, e ao final do treinamento, carregar os pesos do melhor modelo ao invés de se utilizar o último modelo treinado.

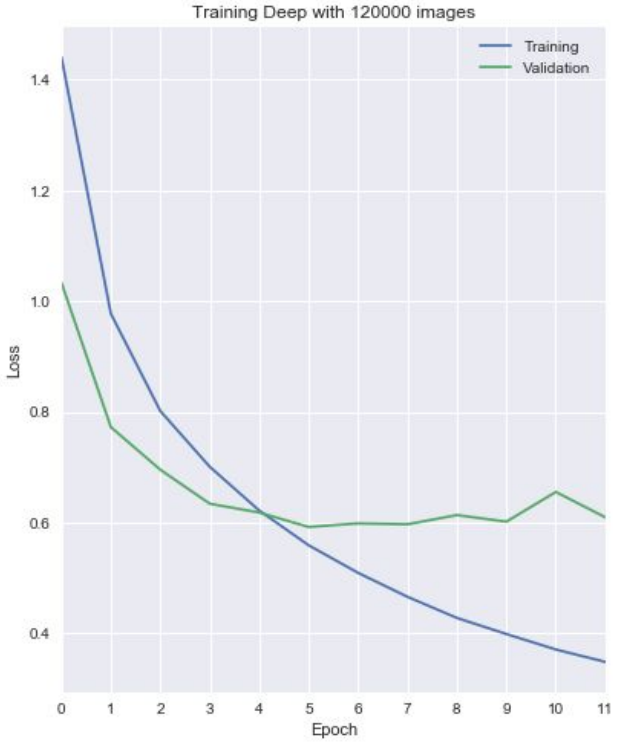
Aumento do conjunto de dados

Outra intuição nesse ponto foi a de que o baixo número de imagens para treinamento estaria limitando a capacidade desses modelos. Nesse caso resolvi utilizar o ImageDataGenerator previamente descrito para aumentar o conjunto de treinamento para 120.000 images, ou seja, gerando duas novas imagens para cada uma preexistente.

Apesar deste processo normalmente ocorrer *on-the-fly*, durante o treinamento do modelo, para acelerar meus experimentos e manter uma consistência para a comparação dos modelos, após a geração do conjunto de dados o mesmo foi salvo para uso posterior.

Com esse novo conjunto de dados 3 vezes maior que o original, é possível perceber uma significativa melhora na performance das arquiteturas testada. Por exemplo, na arquitetura Deep:



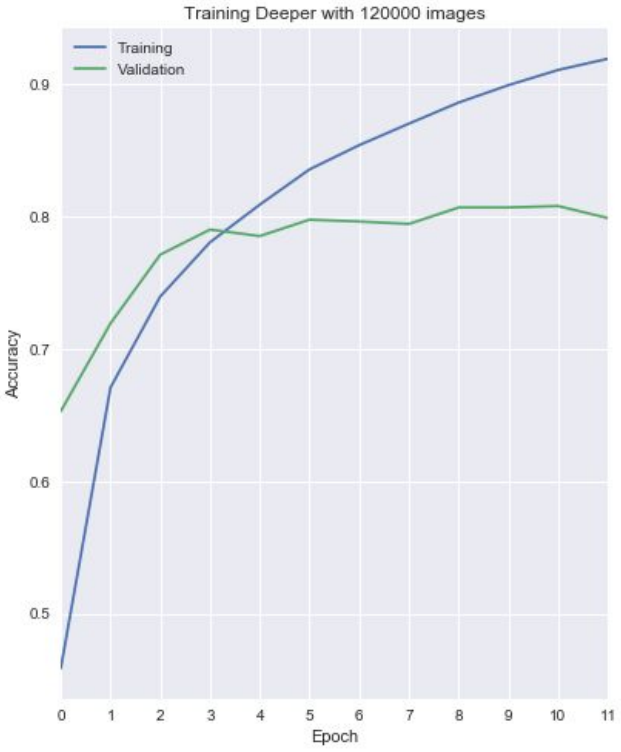


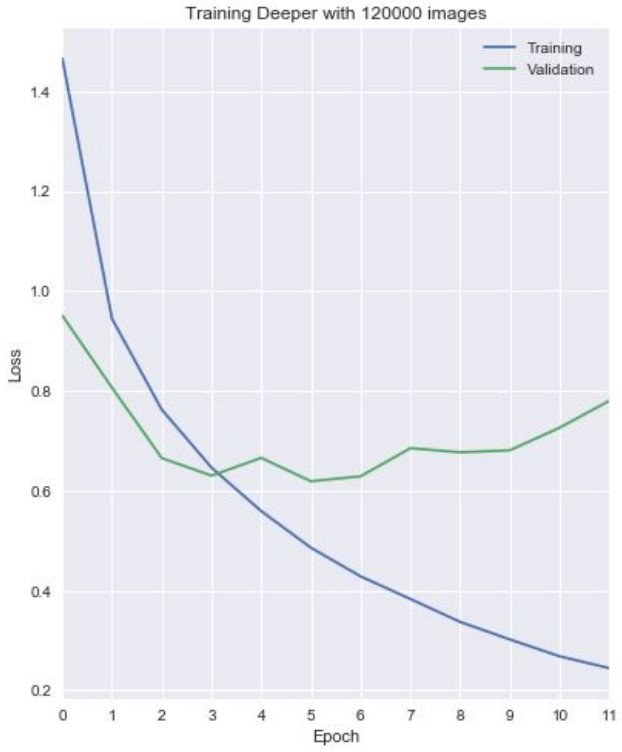
Modelo Deeper

Prosseguindo com a idéia de aprofundar os modelos, criei a arquitetura **deeper**, uma extensão da arquitetura **deep** previamente descrita. Este modelo atingiu o limite de parâmetros treináveis para os 1GB de memória da GPU utilizada, e não consegui modelos mais profundos do que esse.

Tipo de camada	Parâmetros	Entrada	Saída
Convolucional	Kernel 3x3	32x32x3	30x30x64
Convolucional	Kernel 3x3	30x30x64	28x28x64
Max Pooling	Stride 2x2	28x28x64	14x14x64
Convolucional	Kernel 3x3	14x14x64	12x12x128
Convolucional	Kernel 3x3	12x12x128	10x10x128
Max Pooling	Stride 2x2	10x10x128	5x5x128
Convolucional	Kernel 3x3	5x5x128	3x3x256
Convolucional	Kernel 3x3	3x3x256	1x1x256
Flatten		1x1x256	256
Fully Connected		256	256
Fully Connected		256	256
Saída		256	10

Tabela 5 - Modelo Deeper





Infelizmente me parece que a minha técnica de aprofundar o modelo simplesmente adicionando camadas convolucionais atingiu seu limite, mesmo com o novo conjunto de dados, pois não parece haver diferença significativa entre os modelos Deep / Deeper.

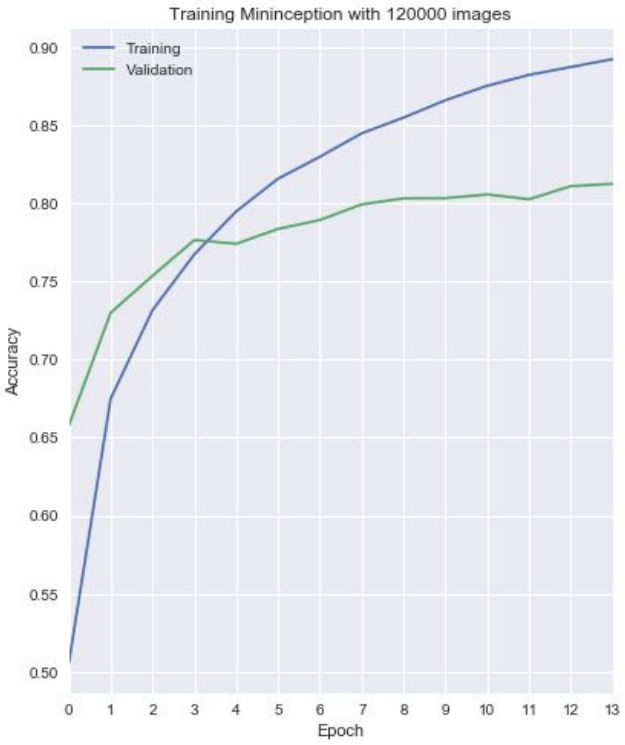
Modelo Mininception

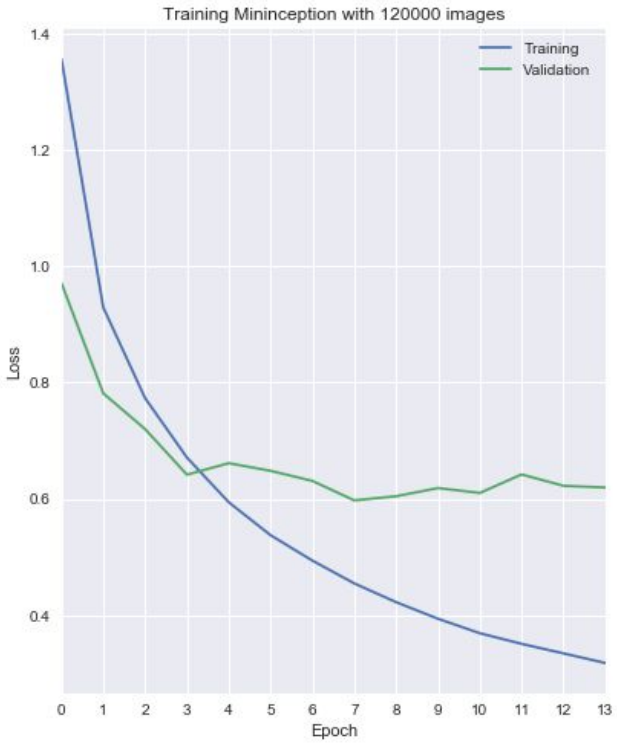
Em busca de modelos alternativos, encontrei a arquitetura Inception já mencionada. Obviamente não consegui utilizá-la a princípio por ser muito mais profunda e necessitar de muito mais memória do que estava disponível. Mas encontrei um conceito muito interessante, onde algumas camadas convolucionais são aplicadas a mesma entrada e a saída de cada uma delas é concatenada, como se fosse uma única camada, um *inception module*.

Em minha implementação cada módulo efetivamente retorna a concatenação de três convoluções: 1x1, 3x3 e 5x5. Implementei uma arquitetura que utiliza 6 desses módulos.

Tipo de camada	Parâmetros	Entrada	Saída
Módulo	Depth 8	32x32x3	32x32x24
Módulo	Depth 16	32x32x24	32x32x48
Max Pool	Stride 2x2	32x32x48	16x16x48
Módulo	Depth 32	16x16x48	16x16x96
Módulo	Depth 64	16x16x96	16x16x192
Max Pool	Stride 2x2	16x16x192	8x8x192
Módulo	Depth 128	8x8x192	8x8x384
Módulo	Depth 256	8x8x384	8x8x768
Max Pool	Stride 2x2	8x8x768	4x4x768
Flatten		4x4x768	12.288
Fully Connected		12.288	384
Dropout	Keep 0.5	384	384
Fully Connected		384	128
Saída		128	10

Tabela 6 - Modelo Mininception





Houve ainda uma pequena melhora com relação aos modelos anteriores, mas ainda nada significativo. Obviamente com a limitação de hardware e de dados de treinamento, aparentemente estou próximo do limite.

Modelo Inception v3

Como uma final tentativa resolvi utilizar esse modelo mas ao invés de treiná-lo por completo, utilizando a técnica de *transfer learning*, que utiliza as camadas do modelo pré-treinado em outro conjunto de dados (ImageNet neste caso) e treina apenas uma pequena quantidade de camadas.

Minha expectativa era de conseguir treinar em GPU as camadas finais do modelo, mas mesmo sem serem treinadas, todas as camadas devem ser carregada na memória. Assim, experimentei fazer um treino na CPU, mas abortei o experimento depois de 33 horas de treinamento.

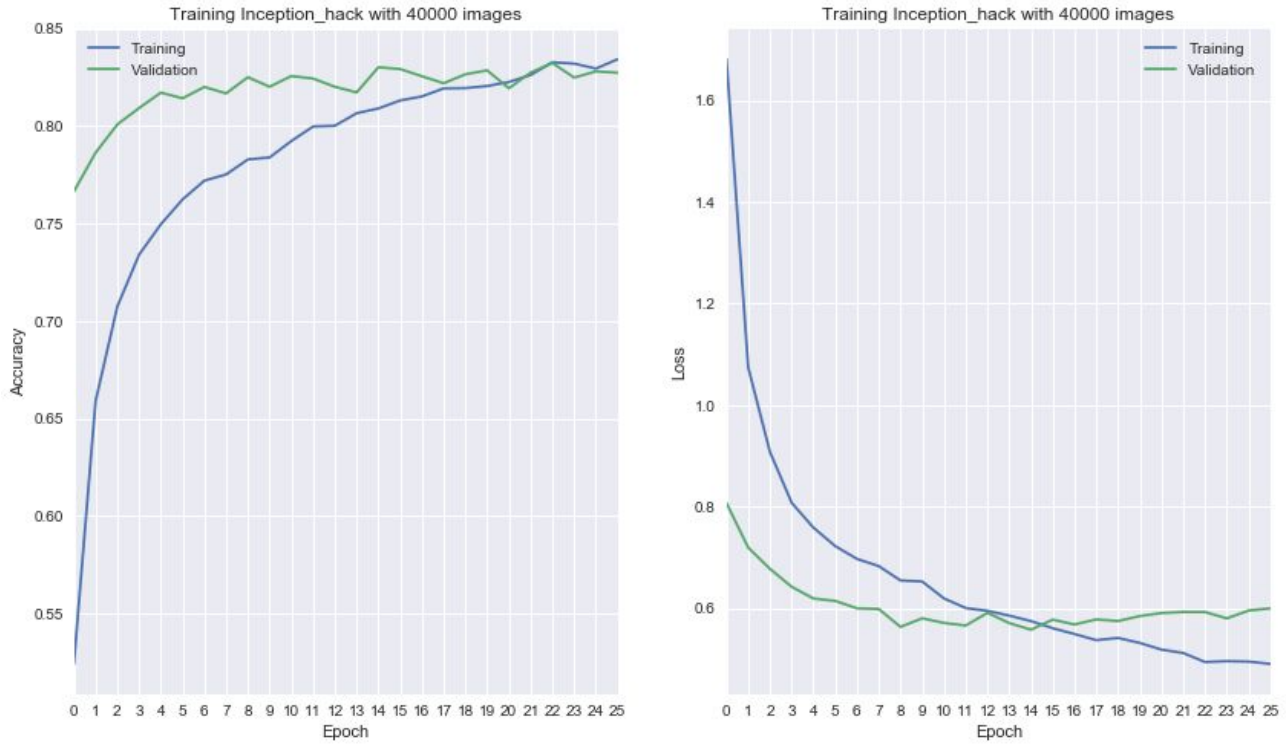
Assim, desenvolvi um pequeno *hack* para conseguir utilizar arquitetura. Minha intuição é de que como a maioria das camadas não são treinadas, sua única função durante o treinamento é fazer a avaliação dos dados de entrada, em uma única etapa de *feedforward* já que não farão parte da etapa de *backpropagation*.

Por isso, dividi essa etapa em duas: primeiramente os dados de entrada são avaliados pela rede Inception mas sem a etapa de classificação e essa avaliação é salva em arquivo. Em seguida, uma rede neural contendo apenas as camadas de classificação é treinada utilizando o resultado da etapa anterior.

A primeira etapa foi efetuada usando a CPU, com duração de 20 horas, mas essa etapa só é necessária uma única vez. Já a segunda etapa, onde o treinamento realmente ocorre, pode ser feita em minutos utilizando-se a GPU por se tratar de um modelo relativamente raso.

Tipo de camada	Parâmetros	Entrada	Saída
Lambda	Resize	32x32x3	139x139x3
Inception v3	Pretrained	139x139x3	3x3x2048
Flatten		3x3x2048	18.432
Fully Connected		18.432	384
Dropout	Keep 0.5	384	384
Fully Connected		384	128
Dropout	Keep 0.5	128	128
Saída		128	10

Tabela 7 - Modelo Inception v3



Devido ao tempo necessário para a primeira etapa do processo, apenas o conjunto de dados original de 40.000 imagens foi usado para o treinamento deste modelo, que na minha opinião tem resultado impressionante, equivalente aos meus modelos mais profundos treinados com 3 vezes mais imagens.

Ainda mais dados de treinamento

Como os modelos parecem ter atingido um teto, e não existindo a possibilidade de modelos maiores devido a restrição computacional, como último experimento treinei todos os modelos, com exceção do Inception v3, com 400.000 imagens.

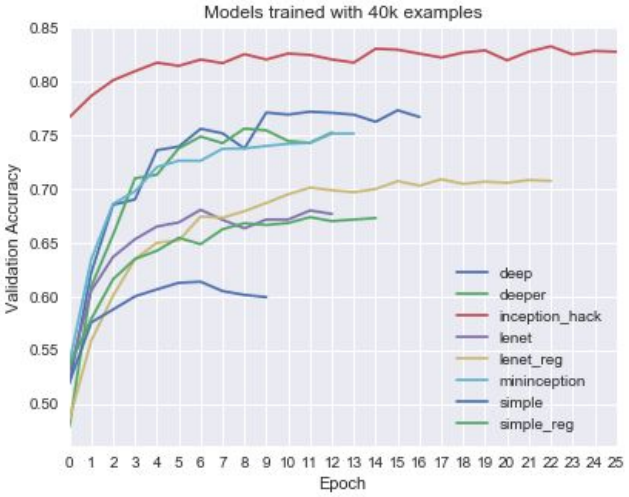
A mesma técnica já descrita foi utilizada, mas desta vez gerando 9 novas imagens para uma presente originalmente. Outra diferença é que desta vez não foi possível salvar esse novo conjunto de dados por limitações do protocolo *Pickle* da linguagem Python pois o arquivo a ser salvo ultrapassa os 12GB.

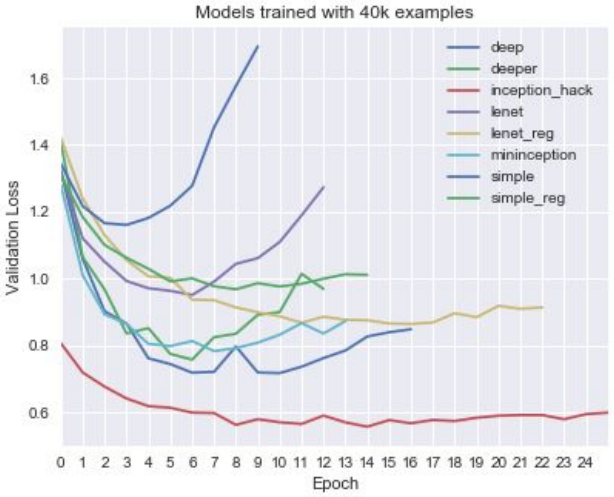
Assim, o treinamento foi feito com as imagens geradas *on-the-fly*.

Comparativo

Abaixo segue um comparativo da evolução do treinamento de todos os modelos, agrupados por tamanho do conjunto de dados de treinamento.

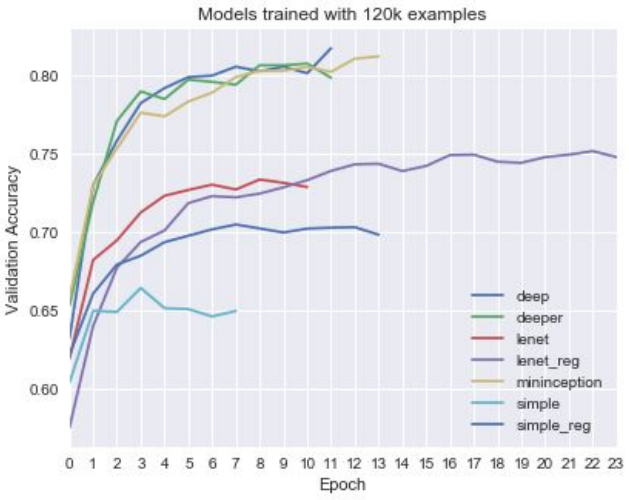
Primeiramente, os modelos treinados com 40.000 imagens:

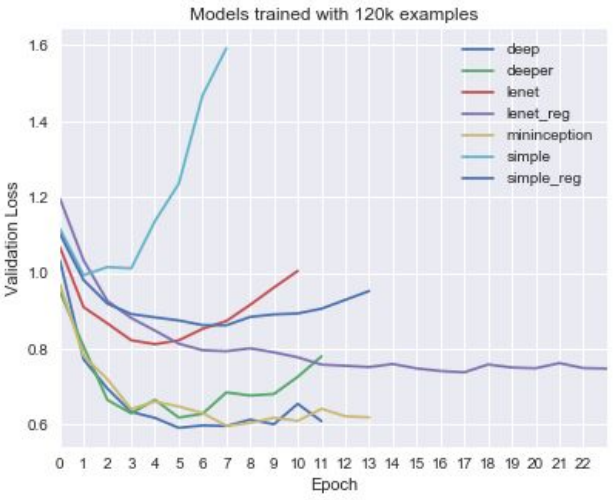




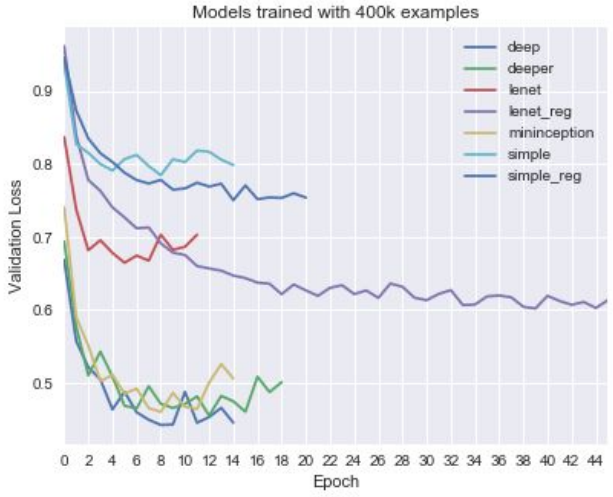
Aqui se torna clara a total supremacia da arquitetura Inception v3, mostrando que a profundidade do modelo realmente é o que torna a técnica de *deep learning* realmente efetiva. Em seguida, os modelos de maior profundidade e por fim os modelos rasos comprovam isso.

Outra coisa interessante nesse gráfico é a diferença de performance entre as versões regularizadas e sem regularização dos modelos Simple e LeNet-5, que comprovam a eficácia das técnicas utilizadas.





Com 120.000 exemplos de treinamento e sem a arquitetura Inception, os modelos profundos se mostram dominantes, mas todos ainda muito próximos, com os modelos rasos vindo em seguida.



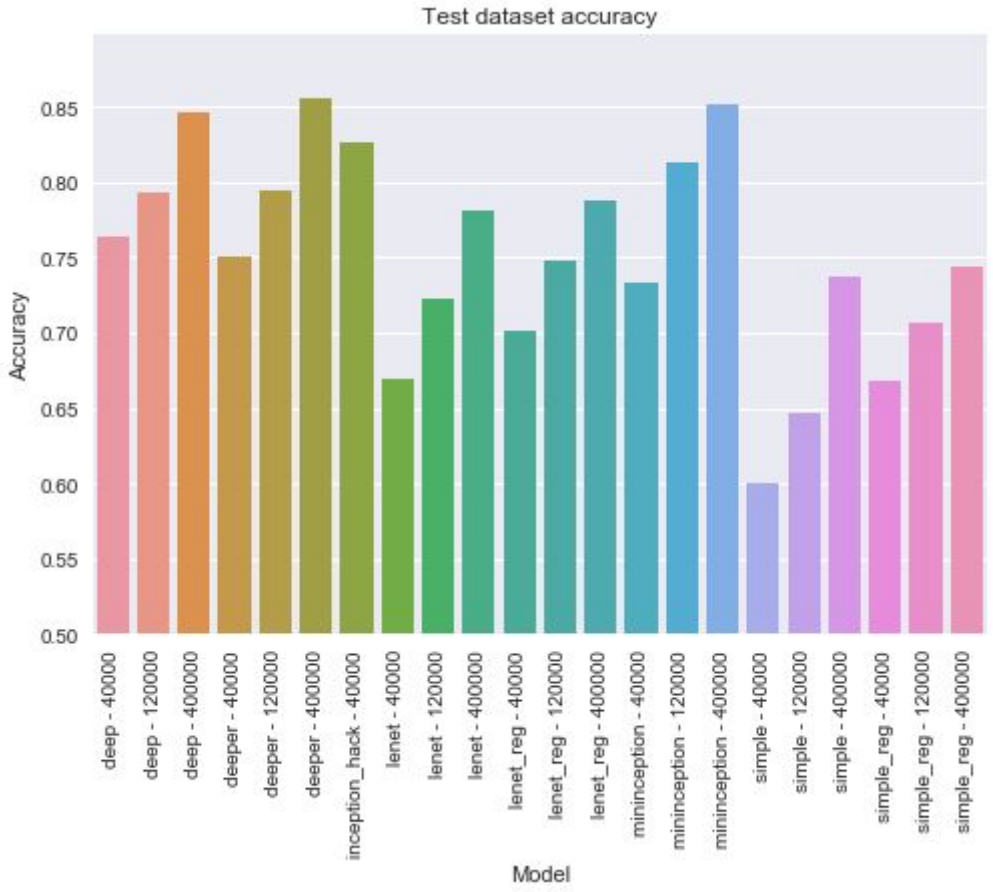
Já com 400.000 exemplos, um fato interessante vem à tona: os modelos regularizados e sem regularização tem desempenho semelhante. Isso obviamente é decorrente da redução da variância através do aumento do *dataset* o que tira a vantagem dos modelos regularizados.

Considerando que os modelos profundos tiveram um excelente ganho de performance ao se multiplicar o tamanho do conjunto de dados por 10, e que a arquitetura Inception conseguiu resultado equivalente com 10 vezes menos dados, gostaria de verificar como se comportaria esse modelo com o conjunto de dados aumentado. Isso poderá ser objeto de um trabalho futuro.

IV. Resultados

Avaliação e validação dos modelos

Com todos os modelos treinados, chega a hora da verdadeira prova, o resultado no conjunto de dados de testes que ainda não foi utilizado. Com isso é possível verificar a real capacidade de generalização dos modelos criados.



Primeiramente é possível confirmar as intuições anteriores com relação à performance dos modelos, com a total supremacia dos modelos mais profundos.

Também é possível perceber o efeito do aumento do número de imagens para treinamento aqui, com um padrão claramente ascendente.

Em seguida, o efeito da regularização nos modelos Simple e LeNet aqui é confirmado, com a redução da eficácia com o aumento no número de imagens.

Outra coisa que fica clara aqui é a robustez do experimento. Ao passo que os conjuntos de dados de 40.000 e 120.000 são exatamente os mesmos para todos os modelos, isso não ocorre com 400.000 imagens, mas ainda assim os mesmos padrões são encontrados nos resultados em todos esses casos.

Justificativa

O objetivo deste projeto nunca foi melhorar os resultados já publicados sobre esse problema, mas é impossível não efetuar uma comparação.

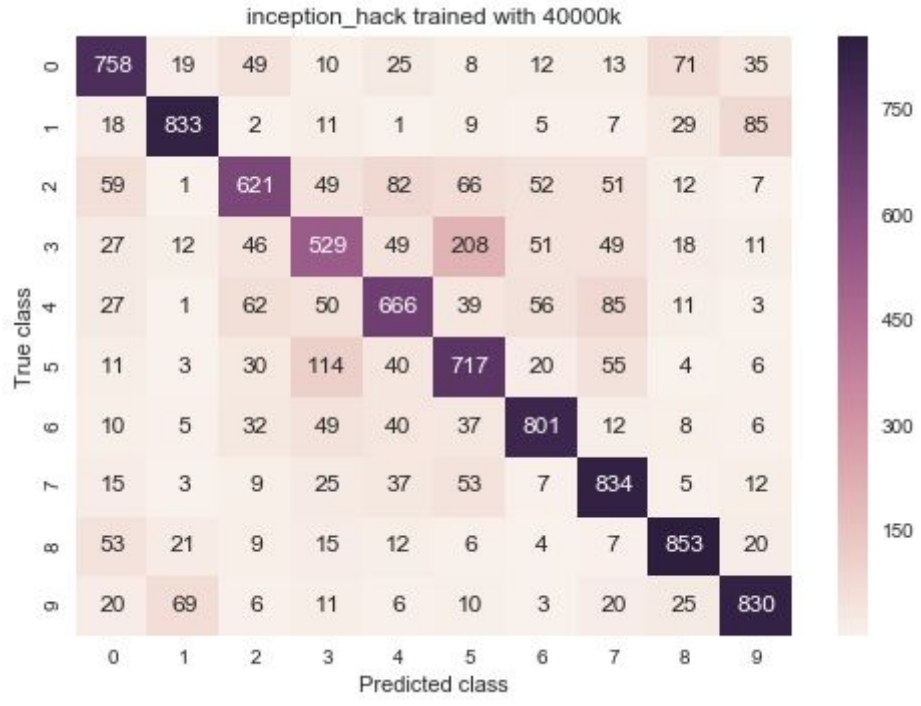
Em relação a uma lista compreensiva de publicações usando esse mesmo conjunto de dados[22], os modelos aqui gerados estão muito aquém dos resultados obtidos nos últimos três anos.

Na verdade esses resultados estão em linha com o estado da arte do ano de 2012. Considerando que a *workstation* utilizada para esse projeto é desta mesma época e sendo esse um projeto de treinamento, fico muito satisfeito com esses resultados.

V. Conclusão

Free-Form Visualization

Uma forma interessante de explorar os resultados é através de uma matriz de confusão. Através desta visualização é possível perceber se há algum viés na classificação efetuada e junto à visualização dos dados de entrada, entender alguns dos problemas enfrentados pelos modelos. Para ilustrar isso, segue a visualização dessa matriz para o modelo Inception.



[22] "Classification datasets results - Rodrigo Benenson." http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html. Acessado em 14 jun. 2017.

É possível perceber que o modelo tem maior dificuldade com as classes 2, 3 e 4. Todas essas classes são referentes à animais, e minha intuição é que o *background* esteja confundindo esse modelo, devido ao ambiente natural desses ser muito parecido. Na minha opinião isso é corroborado pela classificação de mais de 20% das imagens da classe 3 (aves) como classe 5 (sapos).

Reflexão

Acredito que esse projeto compreenda um bom exemplo de como se desenvolve um trabalho utilizando *deep learning*.

Primeiramente, a aquisição dos dados, muito facilitada pelo uso de um conjunto de dados já previamente estabelecido. Em seguida, a exploração desses dados para gerar *insights* sobre o que os modelos gerados deverão enfrentar.

Após uma compreensão do problema, um pré-processamento desses dados pode ser benéfico para melhorar o resultado final do modelo gerado. Em outros casos, pode ser estritamente necessário, como no caso do aumento do tamanho das imagens para se utilizar a arquitetura Inception.

Com o conjunto de dados em mão, abre-se uma bifurcação na estrada: por um lado, a criação de novas arquiteturas através de um processo iterativo ou partindo de arquiteturas já conhecidas, o que é altamente dispendioso e sujeito a erros.

Por outro lado, vem crescendo a utilização não apenas de arquiteturas já estabelecidas, mas também pré-treinadas, que podem rapidamente ser adaptadas para outros usos. Vejo que esse caminho deve ser dominante daqui para frente, pois oferece muita facilidade para a criação de modelos altamente poderosos sem a necessidade de um *know-how* profundo na área. Inclusive, diversos provedores de recursos de nuvem já oferecem serviços incluindo esses modelos pré-treinados[23].

Por fim, antes do uso em produção dos modelos, uma avaliação dos resultados, especialmente da real capacidade de generalização desses se faz necessária.

[23] "How to classify images with TensorFlow using Google Cloud Machine ...." 16 dez. 2016, https://cloud.google.com/blog/big-data/2016/12/how-to-classify-images-with-tensorflow-using-google-cloud-machine-learning-and-cloud-dataflow. Acessado em 14 jun. 2017.

Melhorias

Como já mencionado, nenhuma arquitetura de vanguarda como Inception v4[24] ou ResNet-152[25] foi utilizada aqui, devido a grande capacidade computacional requerida.

Mesmo o experimento com a rede Inception v3 pode ser estendido com o uso do conjunto de dados aumentado, o que também não foi possível nesse projeto.

De qualquer forma, o CIFAR-10 já é considerado um problema resolvido e apenas um benchmark para modelos de classificação de imagens.

Um desafio interessante em desenvolvimento hoje é criar modelos que não apenas identificam quais objetos estão em uma imagem, mas também qual a relação entre eles, como o objeto de criar um real entendimento do que a imagem significa.

[24] "[1602.07261] Inception-v4, Inception-ResNet and the Impact of ...." 23 fev. 2016, https://arxiv.org/abs/1602.07261. Acessado em 14 jun. 2017.
[25] "[1512.03385] Deep Residual Learning for Image Recognition - arXiv.org." 10 dez. 2015, https://arxiv.org/abs/1512.03385. Acessado em 14 jun. 2017.