Rede neural convolutional (convolutional deep learning) em Tiny_DNN

Nesta apostila, vamos estudar rede neural convolucional, usando o material disponível no livro (disponibilizado gratuitamente pelo autor):

http://neuralnetworksanddeeplearning.com/

Esse livro traz os exemplos em Python, usando a biblioteca de deep learning Theano.
No seu lugar, vamos usar a biblioteca de deep learning Tiny-DNN para C++.

Nota 1: Baixei Tiny-DNN em 28/7/2017 a última versão.

https://github.com/tiny-dnn/tiny-dnn

Nota 2: A documentação de tiny-dnn está em:

http://tiny-dnn.readthedocs.io/en/latest/

https://media.readthedocs.org/pdf/tiny-dnn/latest/tiny-dnn.pdf

Deixei esse arquivo em cekeikon5/tiny_dnn/docs

Nota 3: Para compilar um programa.cpp que só usa tiny-dnn, escreva:

```
>compila programa -t
```

Para compilar um programa.cpp que usa tiny-dnn e OpenCV/Cekeikon, escreva:

```
>compila programa -t -c
```

Nota 4: Configurei programa "compila" para compilar tiny-dnn com instruções de vetorização AVX. Se o seu processador não aceita instruções AVX, edite o arquivo:

cekeikon5/tiny_dnn/tiny_dnn/config.h

para desabilitar AVX. Além disso, deve desabilitar AVX ao compilar:

```
>compila programa -t -avx=false
```

Nota 5: No Linux, a compilação é feita usando biblioteca de processamento paralelo TBB da Intel (em Ubuntu/Mint, você deve instalar os pacotes libtbb-dev e libtbb2). No Windows, não consegui usar TBB junto com compilador GCC e TBB será automaticamente desabilitado. Os programas usando Tiny-DNN rodam uns 20-50% mais rápido em Linux do que em Windows, principalmente por usar TBB no Linux.
No Linux, para desabilitar TBB manualmente, edite o arquivo:

cekeikon5/tiny_dnn/tiny_dnn/config.h

e use:

```
>compila programa -t  -tbb=false
```

Nota 6: Tiny_dnn é uma biblioteca “header only”. Assim, a compilação de um programa que usa Tiny_dnn demora bem mais do que um programa que usa apenas Cekeikon/OpenCV, pois deve compilar toda a biblioteca tiny_dnn toda vez que compilar qualquer programa.

Conjunto de treino, validação e teste.

O correto é usar o conjunto teste só no final, quando todo o treino estiver terminado.
O conjunto teste não deve ser usado para verificar a taxa de erro obtida no processo de acerto dos parâmetros. Pode ocorrer overfitting. Deve usar o conjunto de validação para acertar parâmetros.

Importante: Nesta apostila, estamos cometendo este **erro teórico**, pois estamos usando o conjunto de teste para decidir quando para o treino.

Tiny-DNN

Vamos fazer primeiro um teste simples de regressão:

Compilação de programa que usa somente tiny_dnn usando programa compila:

```
>compila regression.cpp -t
```

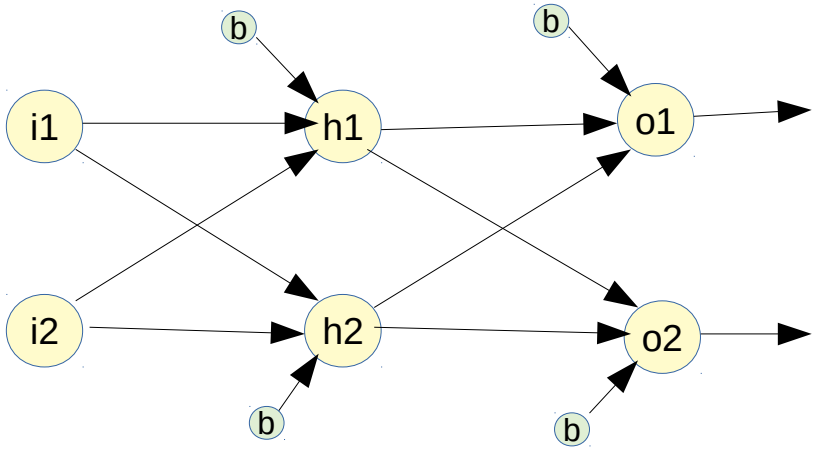
Compilação em Linux sem usar programa compila:

```
>g++ -std=c++14 regression.cpp -o regression -O3 -s -pthread -I/home/usuario/cekeikon5/tiny_dnn -mavx -ltbb
```

Compilação em Windows sem usar programa compila:

```
>g++ -std=c++14 regression.cpp -o regression -O3 -s -Ic:\cekeikon5\tiny_dnn -mavx
```

No meu computador, a compilação demora 14 s.




```
//regression.cpp
#include "tiny_dnn/tiny_dnn.h"
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;

int main() {
  network<sequential> net;
  gradient_descent opt;
  opt.alpha  = 1; // learning rate

  vector<vec_t> ax { {0.9, 0.1}, {0.1, 0.9} };
  vector<vec_t> ay { {0.1, 0.9}, {0.9, 0.1} };

  net << fc(2, 2) << sigmoid()
      << fc(2, 2) << sigmoid();

  int batch_size = 2;
  int epochs = 20000;
  net.fit<mse>(opt, ax, ay, batch_size, epochs);

  vector<vec_t> qx { {0.9, 0.1}, {0.1, 0.9}, {0.8, 0.0}, {0.2, 0.9} };
  for (unsigned i=0; i<qx.size(); i++) {
    vec_t result=net.predict(qx[i]);
    for (unsigned j=0; j<result.size(); j++)
      cout << result[j] << " ";
    cout << endl;
  }
}
```

```
Saída:
0.100003 0.900003
0.899996 0.0999966
0.104157 0.896719
0.881705 0.118153
```
No meu computador, a execução demora 0.265s

Aula 6, exercício 1 (2 pontos): Execute o programa regression.cpp acima. Faça as seguintes alterações:
a) Mude a rede para a estrutura abaixo e grave como regression2.cpp.

```
  net << fc(2, 2) << relu()
      << fc(2, 2);
```

b) Diminua ou aumente learning rate para que a rede convirja mais rapidamente (grave como regression3.cpp).
c) Experimente fazer outras alterações que façam a rede convergir mais rapidamente (em menos epochs). Grave o melhor como regression4.cpp.

Nota: vec_t é o mesmo que vector<float>, mas com alinhamento de memória em 64 bytes para poder usar eficientemente as instruções SIMD (AVX, SSE, etc).

Nota: tensor_t é vector<vec_t>.

As definições desses tipos estão em: `.../tiny_dnn/util/util.h`:

```
typedef std::vector<float_t, aligned_allocator<float_t, 64>> vec_t;
typedef std::vector<vec_t> tensor_t;
```

`net.fit<mse>` é usado para fazer regressão. Neste caso, ay é do tipo vector<vec_t>.

Nota: label_t é o mesmo que size_t (que é algo como "unsigned long int" ou uint64_t). Não sei por que estão usando variável inteira tão grande (64 bits) para representar rótulos (labels).

`net.train<mse>` é usado para fazer classificação. Neste caso, ay é do tipo vector<label_t>.

No lugar de mse, pode-se usar cross_entropy e class_entropy_multiclass.

Vamos fazer primeiro um teste simples de classificação:

```
//classif1.cpp
//compila -t classif1
#include "tiny_dnn/tiny_dnn.h"
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;

int main() {
  network<sequential> net;
  gradient_descent opt;
  opt.alpha  = 1; // learning rate

  vector<vec_t> ax { {0.9, 0.1}, {0.1, 0.9} };
  vector<label_t> ay { 0, 1 };

  net << fc(2, 2) << sigmoid()
      << fc(2, 2) << softmax();

  int batch_size = 2;
  int epochs = 20000;
  net.train<mse>(opt, ax, ay, batch_size, epochs);

  vector<vec_t> qx { {0.9, 0.1}, {0.1, 0.9}, {0.8, 0.0}, {0.2, 0.9} };
  for (unsigned i=0; i<qx.size(); i++) {
    label_t result=net.predict_label(qx[i]);
    cout << result << endl;
  }
}

$ classif1
0
1
0
1
```

Aula 6, exercício 2 (2 pontos): Execute o programa classif1.cpp acima. Faça as seguintes alterações:
a) Mude a rede para a estrutura abaixo e grave como classif2.cpp.

```
  net << fc(2, 2) << relu()
      << fc(2, 2);
```

b) Diminua ou aumente learning rate para que a rede convirja mais rapidamente (grave como classif3.cpp).
c) Experimente fazer outras alterações que façam a rede convergir mais rapidamente (em menos epochs). Grave o melhor como classif4.cpp.

Classificar dígitos MNIST usando rede neural convencional em tiny dnn.

Usando rede com uma única camada escondida.

```
//mlp1.cpp – treina a rede e salva
#include "tiny_dnn/tiny_dnn.h"
#include <string>
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main() {
  int n_train_epochs=2;
  int n_minibatch=10;
  network<sequential> net;
  net << fc(28 * 28, 100) << sigmoid()
      << fc(100, 10) << sigmoid();
  vector<label_t> train_labels;
  vector<vec_t> train_images;
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images,
   -1.0, 1.0, 0, 0);
  adagrad optimizer;
  net.train<mse>(optimizer, train_images, train_labels, n_minibatch,
   n_train_epochs);
  net.save("mlp1.net");
}

//pred1.cpp – carrega a rede e faz predicoes
#include "tiny_dnn/tiny_dnn.h"
#include <string>
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main() {
  network<sequential> net;
  net.load("mlp1.net");
  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images,
   -1.0, 1.0, 0, 0);
  net.test(test_images, test_labels).print_detail(std::cout);
}
```

Nota: -1.0, 1.0 na função parse_mnist_images indica para mapear o pixel mais escuro para -1 e mais claro para +1 durante a leitura.

Depois de treinar 2 epochs:

```
accuracy:92.84% (9284/10000)
    *     0     1     2     3     4     5     6     7     8     9
    0   959     0    13     1     1    10    11     3     5    13
    1     0  1117    10     4     6     5     4    20    13     8
    2     1     1   921    18     3     5     3    20     3     0
    3     1     3    16   930     0    34     1     8    24    10
    4     2     0    16     1   914     6     5     8    11    16
    5     2     0     0    20     0   781    10     1    16     6
    6    10     4    12     3    11    15   921     0    14     1
    7     1     2    13    11     1     6     1   933    12     5
    8     4     8    28    11     4    20     2     1   863     5
    9     0     0     3    11    42    10     0    34    13   945
```

Melhorando a estrutura de rede e treinando durante 30 epochs:

```
//mlp3b.cpp
#include "tiny_dnn/tiny_dnn.h"
#include <string>
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;
string data_dir_path="/home/hae/tiny-dnn-master/data/";

int main() {
  int n_train_epochs=30;
  int n_minibatch=10;
  network<sequential> net;
  net << fc(28 * 28, 400) << sigmoid() << fc(400, 10) << sigmoid();

  vector<label_t> train_labels;
  vector<vec_t> train_images;
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, 0.0, 1.0, 0, 0);
  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images, 0.0, 1.0, 0, 0);

  adagrad optimizer;

  cout << "start training" << endl;
  progress_display disp(train_images.size());
  timer t;

  int epoch = 1;
  // create callback
  auto on_enumerate_epoch = [&]() {
    cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
              << t.elapsed() << "s elapsed." << endl;
    ++epoch;
    result res = net.test(test_images, test_labels);
    cout << res.num_success << "/" << res.num_total << endl;

    disp.restart(train_images.size());
    t.restart();
  };

  auto on_enumerate_minibatch = [&]() { disp += n_minibatch; };

  // training
  net.train<mse>(optimizer, train_images, train_labels, n_minibatch,
                 n_train_epochs, on_enumerate_minibatch, on_enumerate_epoch);
  cout << "end training." << endl;
}
```

Epoch1=92,06%. Demora 66s/epoch.
Epoch30=96,40% (3,6% de erro).

Na apostila redeneural, implementamos do zero rede neural "raso" (com uma única camada escondida) sem usar bibliotecas prontas. O menor erro obtido foi de 2.26% (sem redimensionar as imagens, com nlado=28) e 2.04% (redimensionando para 12x12 pixels, nlado=12). Usamos ativação sigmóide com 100 neurônios na camada escondida. Compare com 3.60% de erro usando tiny-dnn.

Talvez, se selecionarmos valores melhores para a taxa de aprendizagem (alpha), obtenhamos menor taxa de erro.

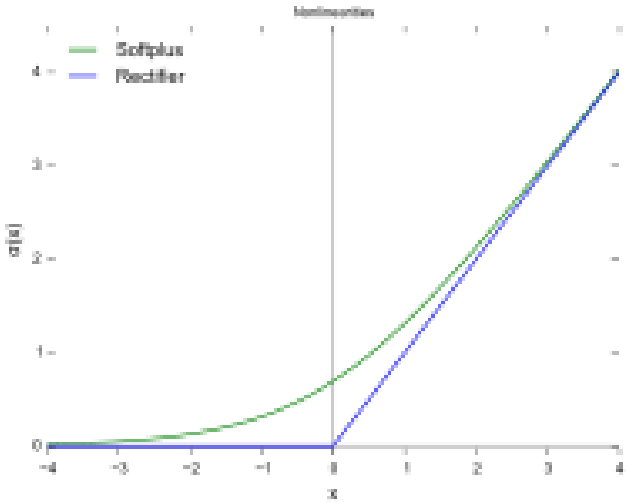
Os dois comandos abaixo (pintados de verde no programa acima) são chamados "funções lambda". Função lambda é uma característica disponível em C++11 e C++14. Função lambda aparece no corpo do programa.

```
 auto on_enumerate_epoch = [&]() {
   (...)
 };

 auto on_enumerate_minibatch = [&]() { disp += n_minibatch; };
```

Abaixo, (quase) o mesmo programa escrito sem usar as funções lambda:

```
//mlp3c.cpp
#include "tiny_dnn/tiny_dnn.h"
#include <string>
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;
string data_dir_path="/home/hae/tiny-dnn-master/data/";

network<sequential> net;
progress_display disp(1);
int n_minibatch=10;
int n_train_epochs=30;
int epoch = 1;
timer t;
vector<label_t> train_labels;
vector<vec_t> train_images;
vector<label_t> test_labels;
vector<vec_t> test_images;

void on_enumerate_epoch() {
  cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
            << t.elapsed() << "s elapsed." << endl;
  ++epoch;
  result res = net.test(test_images, test_labels);
  cout << res.num_success << "/" << res.num_total << endl;
  disp.restart(train_images.size());
  t.restart();
}

void on_enumerate_minibatch() {
  disp += n_minibatch;
}

int main() {
  net << fc(28 * 28, 400) << sigmoid() << fc(400, 10) << sigmoid();
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, 0.0, 1.0, 0, 0);
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images, 0.0, 1.0, 0, 0);

  adagrad optimizer;
  cout << "start training" << endl;
  t.restart();
  disp.restart(train_images.size());
  net.train<mse>(optimizer, train_images, train_labels, n_minibatch,
                 n_train_epochs, on_enumerate_minibatch, on_enumerate_epoch);
  cout << "end training." << endl;
}
```

ReLU (rectified linear unit) é uma função de ativação introduzida no ano 2000 e é a função de ativação mais popular em 2015 [Wikipedia]. Parece a função de transferência vi x vo de um diodo ideal.



Curva relu (rectified linear unit) em azul.

A grande vantagem desta função de ativação é a propagação eficiente de gradiente: não apresenta problema de "vanishing or exploding gradient" (veja mais detalhes no livro). Portanto, é adequado para ser usado em deep learning.

Usando relu.

```
//mlp_relu1.cpp
  network<sequential> net;
  net << fc(28 * 28, 400) << relu()
      << fc(400, 10) << sigmoid();

Epoch1=95,68%. Demora 67s/epoch.
...
Epoch24=98,19%.
Epoch30=98,22% (erro=1,78%)
```

Usando relu, atingimos o menor erro (1,78%) numa rede neural rasa (2 camadas).

A ativação da última camada não pode ser relu:

```
//mlp_relu2.cpp
  network<sequential> net;
  net << fc(28 * 28, 400) << relu()
      << fc(400, 10) << relu();

Epoch1=21,10%. Demora 40s/epoch.
Epoch3=29,77%.
```

Se colocar relu na última camada, a taxa de erro fica muito grande.

Se tirar a função de ativação da última camada, também dá bom resultado.

```
//mlp_relu3.cpp
  network<sequential> net;
  net << fc(28 * 28, 400) << relu()
      << fc(400, 10);
```

```
Epoch 1/30 finished. 27.4309s elapsed. 9627/10000
Epoch 2/30 finished. 27.3367s elapsed. 9672/10000
Epoch 3/30 finished. 27.3066s elapsed. 9705/10000
...
Epoch 29/30 finished. 26.5134s elapsed. 9817/10000
Epoch 30/30 finished. 26.7437s elapsed. 9816/10000
```

Deep neural network é rede neural com muitas camadas escondidas. Um problema para treinar rede neural profunda é "vanishing gradient" ou "exploding gradient". Veja o livro para maiores detalhes. Usando a função de ativação relu, este problema é minimizado.

Vamos inserir mais uma camada escondida (2 camadas escondidas) e ativação relu:

```
//mlp6.cpp
  int n_train_epochs=30;
  int n_minibatch=10;
  network<sequential> net;
  net << fc(28 * 28, 400) << relu()
      << fc(400, 100) << relu()
      << fc(100, 10) << sigmoid();
```

Atingiu 9850 de acerto (erro 1,50%) após epoch 30.
Demorou 50*30 segundos.

Com 2 camadas escondidas e ativação relu, atingimos a menor taxa de erro até agora (1,5%).

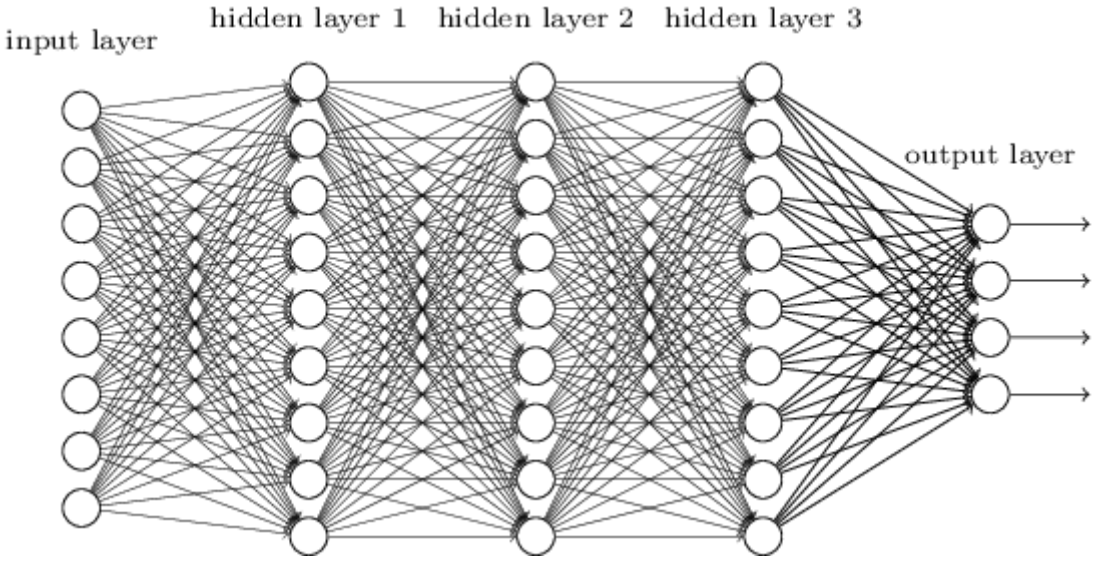
Inserindo mais uma camada escondida (3 camadas escondidas).

```
//mlp7.cpp
  int n_train_epochs=30;
  int n_minibatch=10;
  network<sequential> net;
  net << fc(28 * 28, 400) << relu()
      << fc(400, 200) << relu()
      << fc(200, 100) << relu()
      << fc(100, 10) << sigmoid();
```

Atingiu 9843 de acerto (erro 1,57%) após epoch 30.
Demorou 60*30 segundos.
Conclusão: Não adiantou inserir a terceira camada escondida.

Como diminuir ainda mais a taxa de erro, uma vez que não adianta inserir mais camadas escondidas? Vamos usar *convolutional neural network*. Convolutional neural network é um tipo de rede neural profunda amplamente utilizada no reconhecimento de imagens. As primeiras camadas de CNN consistem de filtros lineares (convoluções).

Numa rede neural convencional, todas as camadas são "fully connected", isto é, todos os neurônios de uma camada estão ligados com todos os neurônios da camada seguinte:

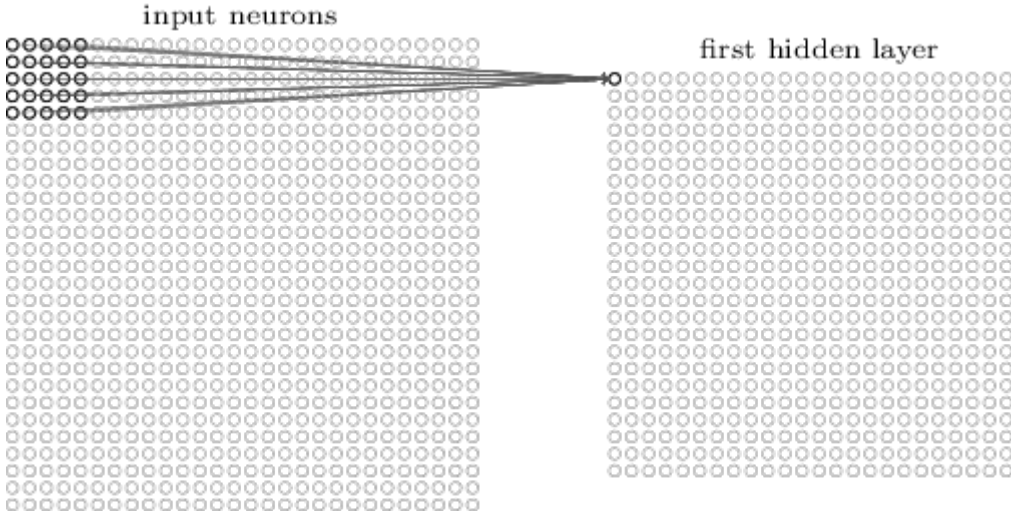


De [http://neuralnetworksanddeeplearning.com]. Rede neural completamente conectado.

Isto é contra a nossa intuição de visão. Não há como a ativação do neurônio no canto superior esquerdo da imagem influenciar a ativação do neurônio da camada seguinte no canto inferior direito.

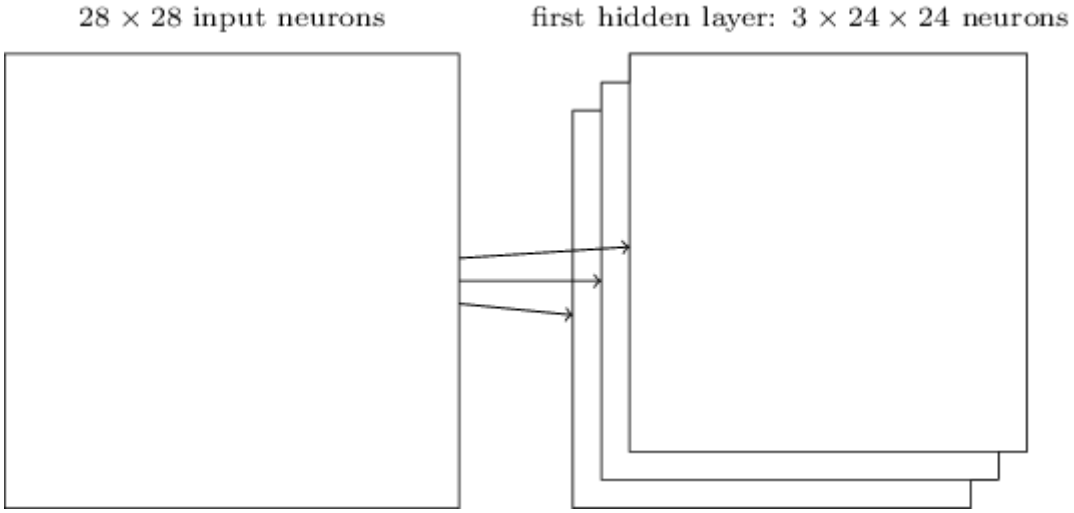
Intuitivamente, como funciona a nossa visão? Primeiro, detectamos as características primitivas, como cantos, arestas, etc. Depois, integramos essas características para detectar formas mais complexas.

As camadas inferiores de CNN serão filtros lineares (convoluções) com bias. Pode-se imaginar que está aplicando vários "template matchings" na imagem, procurando formas simples. Na figura abaixo, um filtro linear 5x5. Os mesmos 25 pesos e bias serão aplicados em todos os pixels, varrendo imagem, como num filtro linear ou convolução ou template matching.



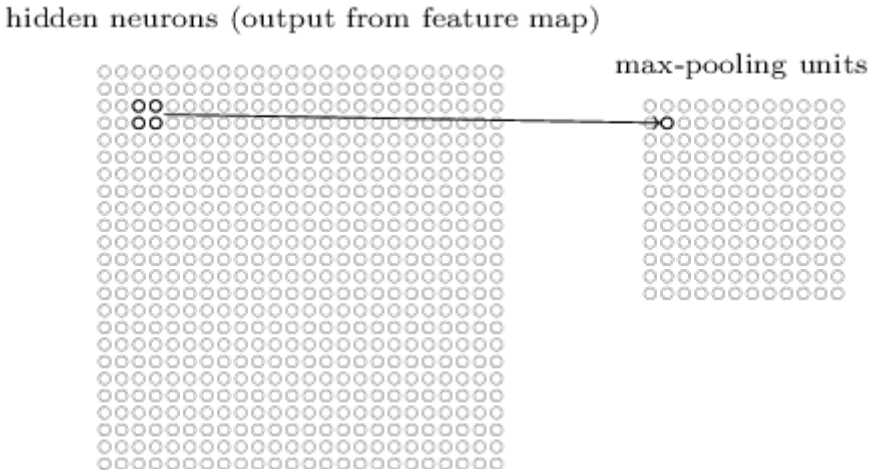
De [http://neuralnetworksanddeeplearning.com]. CNN possui filtro linear nas primeiras camadas.

Vários filtros lineares diferentes serão projetados automaticamente pelo algoritmo de backpropagation. A figura abaixo mostra o projeto e aplicação de 3 filtros diferentes.



De [http://neuralnetworksanddeeplearning.com].

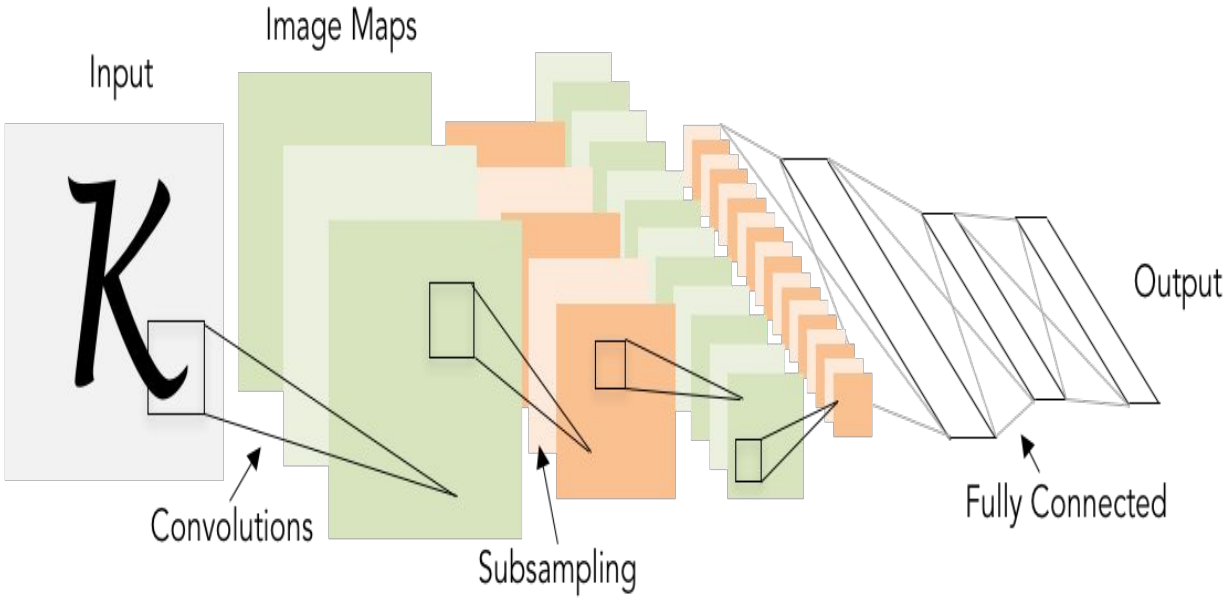
As saídas dos filtros normalmente alimentam camadas tipo "pooling". Estas camadas reduzem a dimensionalidade da imagem, calculando localmente o máximo (max-pool, o mais popular) ou a média (average-pool).



De [http://neuralnetworksanddeeplearning.com].

A saída de camada de agrupamento (pooling) pode entrar em outros filtros lineares. Esta segunda camada convolucional combina as ativações de vários filtros da primeira camada convolucional, procurando descobrir padrões.

As camadas superiores de uma rede CNN são tipicamente camadas convencionais (fully-connected ou inner-product).



Filtros convolucionais são 5x5, aplicados em passo (stride) 1.
Subamostragem (max_pooling) são 2x2, aplicados em passo 2.

Arquitetura é CONV-POOL-CONV-POOL-FC-FC.

Classificar dígitos MNIST usando convolutional neural net em tiny_dnn.

Nota: Parece que o último sigmoide pode ser eliminado. Testar.

`compila cnn0 -t`

```
//cnn0.cpp
#include "tiny_dnn/tiny_dnn.h"
#include <string>
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main() {
  int n_train_epochs=30;
  int n_minibatch=10;

  network<sequential> net;
  net << conv(28, 28, 5, 1, 20) << relu()
      << max_pool(24, 24, 20, 2)
      << fc(12*12*20, 100) << relu()
      << fc(100, 10) << sigmoid();

  vector<label_t> train_labels;
  vector<vec_t> train_images;
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, 0.0, 1.0, 0, 0);
  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images, 0.0, 1.0, 0, 0);

  adagrad optimizer;

  cout << "start training" << endl;
  progress_display disp(train_images.size());
  timer t;

  int epoch = 1;
  auto on_enumerate_epoch = [&]() {
    cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
              << t.elapsed() << "s elapsed." << endl;
    ++epoch;
    result res = net.test(test_images, test_labels);
    cout << res.num_success << "/" << res.num_total << endl;

    disp.restart(train_images.size());
    t.restart();
  };

  auto on_enumerate_minibatch = [&]() { disp += n_minibatch; };

  net.train<mse>(optimizer, train_images, train_labels, n_minibatch,
                 n_train_epochs, on_enumerate_minibatch, on_enumerate_epoch);

  cout << "end training." << endl;
}
```

```
Linux:
Epoch 1/30 finished. 82.1353s elapsed. 9764/10000
Epoch 2/30 finished. 80.5638s elapsed. 9809/10000

Windows:
Epoch 1/30 finished. 99.2199s elapsed. 9764/10000
Epoch 2/30 finished. 99.2399s elapsed. 9809/10000
```

Acerto de 98,77% (erro de 1,23%) após 15 epochs (80 segundos por epoch).

1,23% é o menor erro obtido até agora.

Vamos tentar melhorar colocando mais uma camada de convolução. Teremos 2 camadas de convolução nas camadas inferiores da rede.

```
//cnn4.cpp
#include "tiny_dnn/tiny_dnn.h"
(...)
  int main() {
  int n_train_epochs=30;
  int n_minibatch=10;

  network<sequential> net;
  net << conv(28, 28, 5, 1, 20) << relu()
      << max_pool(24, 24, 20, 2) << relu()
      << conv(12, 12, 5, 20, 40) << relu()
      << max_pool(8, 8, 40, 2) << relu()
      << fc(4*4*40, 1000) << relu()
      << fc(1000, 10) << sigmoid();
  (...)
```

Epoch1=98,67%. Epoch2=98,90%.
Atinge 99,35% (erro=0,65%) de acerto na epoch 30. Demora 30x207s = 103 min.

Não há por que ter função de ativação após camadas de agrupamento (pooling). Retirando essas camadas e pedindo para gravar a rede com a melhor taxa de acerto, obtemos:

```
//cnn_noact2.cpp
#include "tiny_dnn/tiny_dnn.h"
#include <string>
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main() {
  int n_train_epochs=30;
  int n_minibatch=10;

  network<sequential> net;
  net << conv(28, 28, 5, 1, 20)
      << max_pool(24, 24, 20, 2) << relu()
      << conv(12, 12, 5, 20, 40)
      << max_pool(8, 8, 40, 2) << relu()
      << fc(4*4*40, 1000) << relu()
      << fc(1000, 10) << sigmoid();

  vector<label_t> train_labels;
  vector<vec_t> train_images;
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, 0.0, 1.0, 0, 0);
  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images, 0.0, 1.0, 0, 0);

  adagrad optimizer;

  cout << "start training" << endl;
  progress_display disp(train_images.size());
  timer t;

  int epoch = 1;
  int omelhor=-1;
  auto on_enumerate_epoch = [&]() {
    cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
              << t.elapsed() << "s elapsed.";
    ++epoch;
    result res = net.test(test_images, test_labels);
    cout << res.num_success << "/" << res.num_total << endl;

    if (omelhor<res.num_success) {
      omelhor=res.num_success;
      string nomearq="cnn_noact2.net";
      net.save(nomearq);
      cout << "Gravado arquivo " << nomearq << endl;
    }
```

```
    disp.restart(train_images.size());
    t.restart();
  };

  auto on_enumerate_minibatch = [&]() { disp += n_minibatch; };

  net.train<mse>(optimizer, train_images, train_labels, n_minibatch,
                 n_train_epochs, on_enumerate_minibatch, on_enumerate_epoch);

  cout << "end training." << endl;
}
```

```
Epoch1=98,69%.  Demora 150s por epoch sem TBB. Demora 75s por epoch usando TBB.
Epoch2=98,96%
Epoch7=99,23%.
Epoch11=99,37%.
Epoch13=99,41% (erro=0,59%).
Epoch14=99,39%
Epoch30=99,25%
```

Eliminando o último sigmoide (ficou pouquinho melhor):

```
Epoch 1/30 finished. 73.0519s elapsed. 9870/10000
Epoch 2/30 finished. 72.9673s elapsed. 9895/10000
Epoch 3/30 finished. 72.5879s elapsed. 9909/10000
Epoch 4/30 finished. 73.1878s elapsed. 9915/10000
Epoch 5/30 finished. 80.2419s elapsed. 9917/10000
Epoch 6/30 finished. 82.5229s elapsed. 9926/10000
Epoch 7/30 finished. 81.9782s elapsed. 9926/10000
...
```

Esta (0,59%) é a menor taxa de erro observada até agora.

Nota: O programa-exemplo da biblioteca tiny-dnn "train.cpp" atingiu error rate de 1.16% após 30 epochs. Atingiu error rate de 0.98% após 60 epochs (minibatch=16), demorando 40 minutos.

```
$ train --data_path ~/tiny-dnn-master/data --epochs 60
```

Portanto, chegamos numa estrutura de rede melhor do que o exemplo que acompanha a biblioteca tiny-dnn.

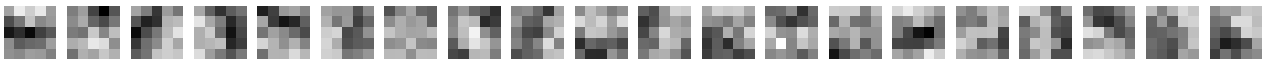
Agora, temos o modelo treinado "cnn_noact2.net" que podemos ler e inspecionar o conteúdo, procurando descobrir como funciona uma rede neural convolucional.

O programa abaixo imprime a estrutura da rede, os pesos das camadas convolucionais como imagens e os pesos e bias da primeira camada convolucional como números.

```
//inspnet1.cpp
//compila inspnet1 -c -t
#include <cektiny.h>
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main() {
  network<sequential> net;
  net.load("cnn_noact2.net");
  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images,
    0.0, 1.0, 0, 0);
  net.test(test_images, test_labels).print_detail(cout);
  cout << endl;

  //imprime a estrutura da rede
  for (int i=0; i<net.depth(); i++) {
    using std::operator<<;
    cout << "#layer: " << i << "\n";
    cout << "layer type: " << net[i]->layer_type() << "\n";
    cout << "input: " << net[i]->in_data_size() << "(" << net[i]->in_data_shape() << ")\n";
    cout << "output: " << net[i]->out_data_size() << "(" << net[i]->out_data_shape() << ")\n";
  }
  cout << endl;

  //imprime os pesos das duas camadas convolucionais como imagens
  image<> img1 = net.at<conv>(0).weight_to_image();
  img1.write("kernel0.bmp");
  image<> img2 = net.at<conv>(3).weight_to_image();
  img2.write("kernel3.bmp");

  //imprime o vetor de pesos da primeira camada convolucional
  vector<vec_t*> weights = net[0]->weights();
  xprint(net[0]->weights().size());
  for (unsigned l=0; l<weights.size(); l++) {
    vec_t& v=(*weights[l]);
    for (unsigned c=0; c<v.size(); c++)
      printf("%+6.3f ",v[c]);
    printf("\n");
  }
}
```

Saída:

```
accuracy:99.41% (9941/10000)
    *     0     1     2     3     4     5     6     7     8     9
    0   975     0     0     0     0     1     1     0     3     0
    1     1  1135     1     0     0     0     3     3     0     0
    2     0     0  1027     1     1     0     0     0     2     0
    3     0     0     1  1005     0     4     0     0     1     0
    4     0     0     0     0   979     0     1     0     0     5
    5     0     0     0     3     0   885     1     0     2     3
    6     1     0     0     0     0     1   951     0     0     0
    7     2     0     3     0     0     0     0  1023     1     2
    8     1     0     0     1     0     0     1     0   963     1
    9     0     0     0     0     2     1     0     2     2   998

#layer: 0
layer type: conv
input: 784([[28x28x1]])
output: 11520([[24x24x20]])
#layer: 1
layer type: relu-activation
input: 11520([[24x24x20]])
output: 11520([[24x24x20]])
#layer: 2
layer type: max-pool
input: 11520([[24x24x20]])
output: 2880([[12x12x20]])
#layer: 3
layer type: conv
input: 2880([[12x12x20]])
output: 2560([[8x8x40]])
#layer: 4
layer type: relu-activation
input: 2560([[8x8x40]])
output: 2560([[8x8x40]])
#layer: 5
layer type: max-pool
input: 2560([[8x8x40]])
output: 640([[4x4x40]])
#layer: 6
layer type: fully-connected
input: 640([[640x1x1]])
output: 1000([[1000x1x1]])
#layer: 7
layer type: relu-activation
input: 1000([[1000x1x1]])
output: 1000([[1000x1x1]])
#layer: 8
layer type: fully-connected
input: 1000([[1000x1x1]])
output: 10([[10x1x1]])
#layer: 9
layer type: sigmoid-activation
input: 10([[10x1x1]])
output: 10([[10x1x1]])

net[0]->weights().size() = 2  [pesos e bias]
+0.054 +0.212 +0.146 +0.203 +0.042 -0.127 -0.040 +0.124 +0.000 +0.033 -0.244 -0.195 -0.270 -0.154 -0.102 +0.009 -0.094
-0.068 -0.058 +0.045 +0.018 +0.000 +0.046 +0.038 -0.018 -0.023 +0.046 -0.148 -0.291 -0.110 +0.006 +0.071 -0.076 -0.132
-0.036 -0.028 -0.048 +0.137 -0.004 +0.113 +0.078 +0.103 +0.202 +0.179 +0.013 -0.052 +0.084 +0.056 +0.133 +0.085 +0.058
-0.024 -0.174 +0.042 -0.042 -0.108 -0.219 -0.187 +0.091 +0.108 -0.269 -0.129 -0.034 +0.105 +0.193 -0.172 -0.054 +0.006
+0.150 +0.064 -0.094 -0.046 +0.125 +0.148 +0.130 +0.066 -0.084 -0.184 -0.093 -0.048 +0.193 +0.064 -0.023 -0.206 -0.115
+0.081 +0.142 -0.062 -0.251 -0.139 +0.082 +0.097 +0.116 -0.208 -0.160 +0.178 +0.043 -0.033 -0.114 -0.033 -0.224 -0.155
+0.034 +0.058 -0.008 +0.062 -0.165 -0.246 -0.208 -0.087 -0.043 +0.052 +0.004 -0.064 -0.133 +0.162 +0.093 +0.017 +0.086
-0.012 -0.054 +0.090 +0.088 +0.173 +0.100 +0.055 +0.028 +0.032 -0.115 +0.008 +0.087 -0.039 +0.037 -0.157 -0.072 +0.145
+0.162 -0.135 -0.101 +0.020 +0.163 +0.077 -0.121 -0.095 -0.085 +0.152 +0.138 -0.040 -0.018 +0.028 -0.116 -0.007 +0.061
-0.069 -0.085 +0.008 +0.079 +0.036 +0.059 -0.086 +0.127 +0.080 +0.058 +0.063 +0.107 +0.011 +0.029 +0.126 +0.003 -0.005
+0.104 +0.106 +0.013 +0.108 +0.104 -0.043 +0.029 -0.068 -0.112 -0.193 -0.008 +0.129 +0.105 -0.205 -0.165 +0.007 +0.115
+0.226 +0.013 -0.084 -0.098 +0.080 +0.195 +0.134 +0.027 -0.137 -0.159 +0.097 +0.105 -0.025 -0.008 +0.008 -0.066 -0.175
+0.053 +0.036 -0.084 -0.100 +0.065 +0.086 -0.034 -0.136 -0.154 +0.001 +0.153 -0.012 -0.090 -0.004 +0.180 -0.015 -0.126
-0.084 -0.062 +0.110 +0.125 +0.004 +0.114 +0.056 -0.026 +0.016 +0.170 +0.108 +0.161 +0.049 +0.174 -0.030 +0.129 +0.118
+0.034 -0.017 -0.196 -0.070 -0.102 -0.085 -0.142 -0.065 -0.200 -0.203 +0.027 -0.047 -0.085 -0.125 +0.031 +0.154 +0.142
-0.183 -0.116 +0.025 +0.062 +0.024 -0.140 -0.008 +0.169 +0.076 +0.060 -0.064 -0.074 +0.128 +0.101 -0.074 -0.034 +0.069
+0.083 +0.013 -0.113 +0.058 +0.151 +0.136 +0.144 +0.067 +0.019 +0.006 -0.035 +0.061 -0.013 -0.140 -0.105 +0.082 -0.036
+0.160 -0.181 -0.092 -0.197 -0.015 +0.062 -0.160 -0.087 -0.080 -0.160 -0.124 -0.071 -0.062 +0.007 -0.185 -0.066 +0.002
-0.119 -0.178 -0.117 -0.080 +0.158 +0.023 +0.009 +0.177 +0.079 +0.068 +0.251 -0.008 +0.128 +0.068 -0.003 +0.029 -0.006
-0.098 -0.016 +0.141 +0.135 +0.140 -0.019 +0.112 -0.069 +0.093 -0.049 -0.058 +0.050 +0.066 +0.048 +0.095 +0.021 +0.021
-0.116 -0.032 +0.090 -0.102 +0.019 -0.248 -0.135 -0.059 -0.017 -0.063 +0.163 +0.210 +0.152 +0.106 +0.034 -0.017 -0.001
+0.109 +0.017 -0.059 -0.017 -0.138 -0.246 -0.285 -0.062 -0.142 -0.152 -0.154 -0.103 -0.028 +0.061 +0.006 +0.054 +0.214
+0.145 +0.018 -0.025 +0.052 +0.113 +0.053 -0.043 -0.022 -0.057 +0.110 +0.066 +0.043 +0.121 +0.115 +0.122 -0.145 +0.155
+0.106 -0.091 -0.051 -0.160 +0.082 +0.023 +0.011 +0.045 -0.077 +0.170 +0.161 +0.010 -0.110 +0.042 -0.091 +0.087 +0.127
-0.090 -0.129 -0.031 +0.006 +0.113 -0.122 -0.140 -0.047 +0.132 +0.111 +0.032 -0.202 -0.064 +0.043 +0.123 -0.124 -0.124
-0.056 -0.136 -0.128 -0.037 +0.043 +0.026 -0.173 -0.200 -0.125 -0.009 +0.188 +0.159 -0.030 -0.133 -0.074 +0.015 +0.034
+0.093 +0.064 -0.057 +0.162 +0.173 +0.136 +0.097 +0.037 +0.107 -0.019 +0.086 +0.172 +0.129 -0.077 +0.039 -0.096 +0.143
+0.155 -0.084 -0.079 -0.046 -0.019 +0.116 -0.058 -0.089 -0.124 -0.014 +0.129 -0.069 -0.097 -0.139 +0.060 +0.034 +0.007
+0.055 +0.047 +0.132 +0.123 -0.052 -0.037 +0.175 +0.175 +0.109 -0.076 -0.132 -0.062 +0.157 +0.134 -0.112 -0.231 -0.195
-0.128 +0.060 -0.160 -0.047 -0.127 -0.020 +0.006
-0.000 +0.005 +0.008 +0.018 +0.021 -0.000 -0.000 -0.000 +0.005 +0.011 -0.000 +0.009 +0.008 +0.007 +0.010 -0.007 +0.000
+0.019 -0.042 +0.000
```



kernel0.bmp. Os níveis de cinza das 20 sub-imagens são os pesos que aparecem na listagem acima. Estas são os 20 filtros lineares da primeira camada que a rede neural convolucional utiliza. Nota: Branco e preto estão invertidos.

Invertendo branco com preto.

Convolutional neural net funciona como se calculasse template matching de template matching de template matching ...

```
//cnn_noact2.cpp - copiado para facilitar acompanhar a explicação.
  net << conv(28, 28, 5, 1, 20) << relu()   //camadas 0 e 1
      << max_pool(24, 24, 20, 2)            //camada 2
      << conv(12, 12, 5, 20, 40) << relu()  //camadas 3 e 4
      << max_pool(8, 8, 40, 2)              //camada 5
      << fc(4*4*40, 1000) << relu()         //camadas 6 e 7
      << fc(1000, 10) << sigmoid();         //camadas 8 e 9
```

Camadas 0 e 1: Há 28x28 neurônios de entrada. A camada 1 recebe 28x28 níveis de cinzas, aplica os 20 filtros lineares 5x5 acima, e gera 20 imagens 24x24 como saída. Executa ativação relu na saída.

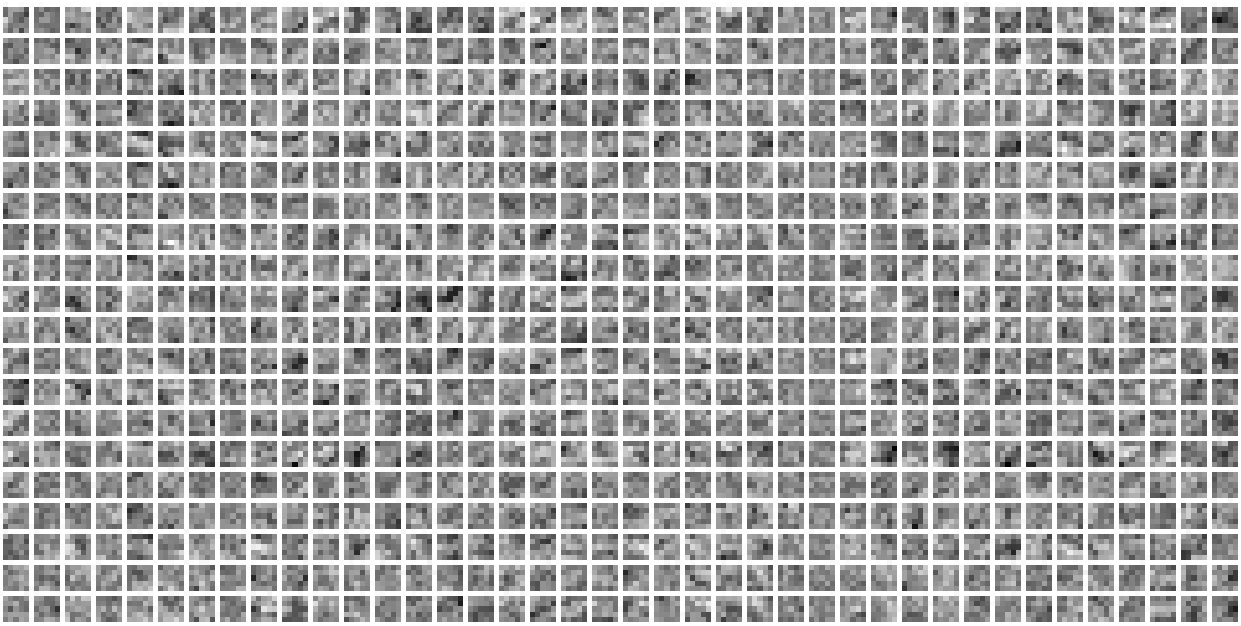
Camada 2: Recebe 20 imagens 24x24, executa max_pool, e gera 20 imagens 12x12 na saída.

Camadas 3 e 4: Recebe 20 imagens 12x12. Cria 40 filtros lineares, onde cada filtro linear está conectado com 20x5x5 pixels (portanto, tem 20x5x5 pesos). Gera 40 imagens 8x8 na saída. Executa ativação relu.

Camada 5: Recebe 40 imagens 8x8 e aplica max_pool, para gerar 40 imagens 4x4 na saída.

Camadas 6 e 7: Recebe 40 imagens 4x4 que estão completamente conectadas com 1000 neurônios. Faz ativação relu.

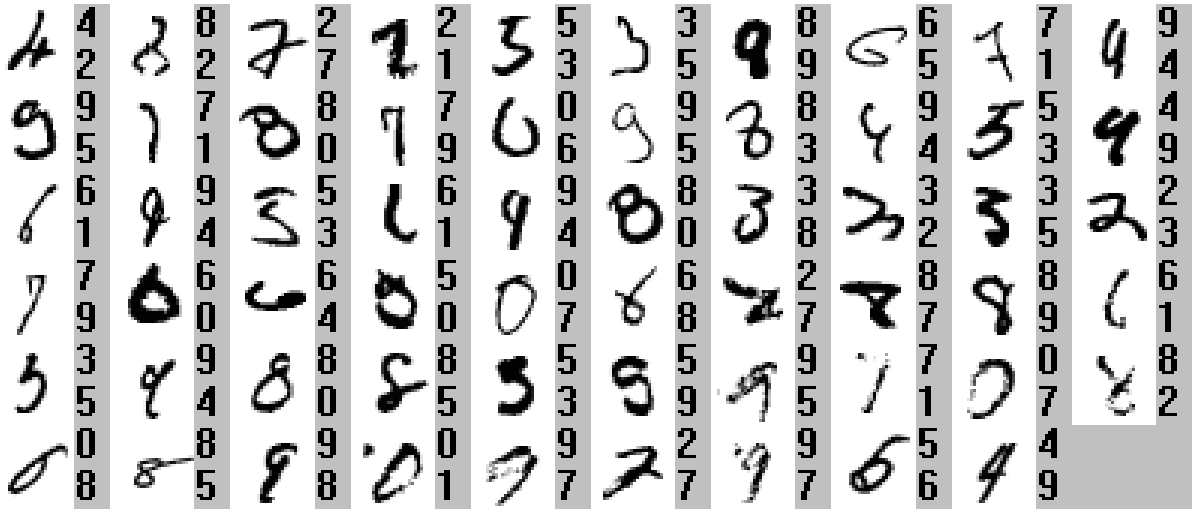
Camada 8: Recebe 1000 entradas e conecta completamente a 10 saídas. Faz ativação sigmoide.



kernel3.bmp. Estes são os 40 filtros 20x5x5 da segunda camada.

O programa abaixo imprime os dígitos classificados incorretamente.

```
//geraerro2.cpp
//compila geraerro2 -c -t
#include <cektiny.h>
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

Mat_<GRY> geraSaida(const vec_t& qx, label_t qy, label_t qp) {
  Mat_<GRY> q; converte(qx,q,28,28); q=255-q;
  Mat_<GRY> d(28,38,192);
  putTxt(d,0,28,to_string(qy));
  putTxt(d,14,28,to_string(qp));
  copia(q,d,0,0);
  return d;
}

Mat_<GRY> geraSaidaErros(const vector<vec_t>& qx,
  const vector<label_t>& qy, const vector<label_t>& qp,
  int nl, int nc) {
  // Gera uma imagem com os primeiros nl*nc digitos classificados erradamente
  Mat_<GRY> e(28*nl,40*nc,192);
  int j=0;
  for (int l=0; l<nl; l++)
    for (int c=0; c<nc; c++) {
      //acha o proximo erro
      while (qp[j]==qy[j] && j<qp.size()) j++;
      if (j==qp.size()) goto saida;
      Mat_<GRY> t=geraSaida(qx[j],qy[j],qp[j]);
      copia(t,e,28*l,40*c);
      j++;
    }
  saida:
  return e;
}

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("Geraerro2: Gera imagem com digitos classificados incorretamente\n");
    erro("Geraerro2 rede.net erros.png");
  }
  network<sequential> net;
  net.load(argv[1]);
  vector<label_t> qy;
  vector<vec_t> qx;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &qy);

  //Escolha uma das duas linhas abaixo.
  //parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &qx, -1.0, 1.0, 0, 0);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &qx, 0.0, 1.0, 0, 0);

  assert(qx.size()==qy.size());
  vector<label_t> qp(qy.size());

  //Faz predicao
  for (unsigned i=0; i<qx.size(); i++)
    qp[i]=net.predict_label(qx[i]);

  int erros=0;
  for (unsigned i=0; i<qx.size(); i++)
    if (qy[i]!=qp[i]) erros++;
  printf("Numero de erros=%d\n",erros);

  //Gera imagem de caracteres reconhecidos erradamente
  Mat_<GRY> e=geraSaidaErros(qx,qy,qp,erros/10+1,10);
  imp(e,argv[2]);
}
```



Os 59 caracteres classificados incorretamente. Superior direito - classificação verdadeira. Inferior direito - classificação atribuída pelo algoritmo. Repare que a rede ainda errou várias imagens que um ser humano classificaria sem problema.

Para diminuir ainda mais a taxa de erro, vamos aumentar a base de dados de treino, deslocando um pixel em cada direção (norte, sul, leste, oeste). Com isso, a base de treino torna-se cinco vezes a base original.

Aqui, vamos usar o header "cektiny.h", que possui as funções de conversão (converte) entre vec_t do tiny-dnn e Mat_<FLT>, Mat_<GRY> e Mat_<COR> de OpenCV. Também possui a função para aumentar o número de amostras de treinamento.

Este arquivo já está incluída na biblioteca Cekeikon.

```
//cektiny.h
#include <cekeikon.h>
#include <tiny_dnn/tiny_dnn.h>
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;

void converte(const vec_t& v, Mat_<FLT>& m, int nl, int nc) {
  if (int(v.size())!=nl*nc) xerro1("Erro converte: Dimensoes nl nc invalidas");
  m.create(nl,nc);
  for (unsigned i=0; i<m.total(); i++)
    m(i)=v[i];
}

void converte(const Mat_<FLT>& m, vec_t& v) {
  v.resize(m.total());
  for (unsigned i=0; i<m.total(); i++) {
    v[i]=m(i);
  }
}

void converte(const vec_t& v, Mat_<GRY>& m, int nl, int nc) {
  if (int(v.size())!=nl*nc) xerro1("Erro converte: Dimensoes nl nc invalidas");
  m.create(nl,nc);
  for (unsigned i=0; i<m.total(); i++)
    m(i)=F2G(v[i]);
}

void converte(const Mat_<GRY>& m, vec_t& v) {
  v.resize(m.total());
  for (unsigned i=0; i<m.total(); i++) {
    v[i]=G2F(m(i));
  }
}

void converte(const vec_t& v, Mat_<COR>& m, int nl, int nc) {
  if (int(v.size())!=3*nl*nc) xerro1("Erro converte: Dimensoes nl nc invalidas");
  m.create(nl,nc);
  int j=0;
  for (int p=0; p<3; p++)
    for (unsigned i=0; i<m.total(); i++)
      m(i)[2-p]=F2G(v[j++]);
}

void converte(const Mat_<COR>& m, vec_t& v) {
  v.resize(3*m.total());
  int j=0;
  for (int p=0; p<3; p++)
    for (unsigned i=0; i<m.total(); i++)
      v[j++]=G2F(m(i)[2-p]);
}

void converte(const vec_t& v, Mat_<CORF>& m, int nl, int nc) {
  if (int(v.size())!=3*nl*nc) xerro1("Erro converte: Dimensoes nl nc invalidas");
  m.create(nl,nc);
  int j=0;
  for (int p=0; p<3; p++)
    for (unsigned i=0; i<m.total(); i++)
      m(i)[2-p]=v[j++];
}

void converte(const Mat_<CORF>& m, vec_t& v) {
  v.resize(3*m.total());
  int j=0;
  for (int p=0; p<3; p++)
    for (unsigned i=0; i<m.total(); i++)
      v[j++]=m(i)[2-p];
```

```
}

vec_t deslocaEsquerda(const vec_t& v, int nl, int nc, int ns) {
  //ns = numero de slices ou numero de espectros ou numero de cores
  //Neste momento, ns==1 ou ns==3
  if (v.size()!=unsigned(nl*nc*ns)) xerro1("Erro: Dimensao invalida");
  if (ns==3) {
    Mat_<CORF> f; converte(v,f,nl,nc);
    for (int c=0; c<f.cols-1; c++)
      for (int l=0; l<f.rows; l++)
        f(l,c)=f(l,c+1);
    vec_t w; converte(f,w);
    return w;
  } else if (ns==1) {
    Mat_<FLT> f; converte(v,f,nl,nc);
    for (int c=0; c<f.cols-1; c++)
      for (int l=0; l<f.rows; l++)
        f(l,c)=f(l,c+1);
    vec_t w; converte(f,w);
    return w;
  } else xerro1("Erro deslocaEsquerda: deve ter ns==1 ou ns==3");
}

vec_t deslocaDireita(const vec_t& v, int nl, int nc, int ns) {
  //ns = numero de slices ou numero de espectros ou numero de cores
  //Neste momento, ns==1 ou ns==3
  if (v.size()!=unsigned(nl*nc*ns)) xerro1("Erro: Dimensao invalida");
  if (ns==3) {
    Mat_<CORF> f; converte(v,f,nl,nc);
    for (int c=f.cols-1; c>0; c--)
      for (int l=0; l<f.rows; l++)
        f(l,c)=f(l,c-1);
    vec_t w; converte(f,w);
    return w;
  } else if (ns==1) {
    Mat_<FLT> f; converte(v,f,nl,nc);
    for (int c=f.cols-1; c>0; c--)
      for (int l=0; l<f.rows; l++)
        f(l,c)=f(l,c-1);
    vec_t w; converte(f,w);
    return w;
  } else xerro1("Erro deslocaDireita: deve ter ns==1 ou ns==3");
}

vec_t deslocaCima(const vec_t& v, int nl, int nc, int ns) {
  //ns = numero de slices ou numero de espectros ou numero de cores
  //Neste momento, ns==1 ou ns==3
  if (v.size()!=unsigned(nl*nc*ns)) xerro1("Erro: Dimensao invalida");
  if (ns==3) {
    Mat_<CORF> f; converte(v,f,nl,nc);
    for (int l=0; l<f.rows-1; l++)
      for (int c=0; c<f.cols; c++)
        f(l,c)=f(l+1,c);
    vec_t w; converte(f,w);
    return w;
  } else if (ns==1) {
    Mat_<FLT> f; converte(v,f,nl,nc);
    for (int l=0; l<f.rows-1; l++)
      for (int c=0; c<f.cols; c++)
        f(l,c)=f(l+1,c);
    vec_t w; converte(f,w);
    return w;
  } else xerro1("Erro deslocaCima: deve ter ns==1 ou ns==3");
}

vec_t deslocaBaixo(const vec_t& v, int nl, int nc, int ns) {
  //ns = numero de slices ou numero de espectros ou numero de cores
  //Neste momento, ns==1 ou ns==3
  if (v.size()!=unsigned(nl*nc*ns)) xerro1("Erro: Dimensao invalida");
  if (ns==3) {
    Mat_<CORF> f; converte(v,f,nl,nc);
    for (int l=f.rows-1; l>0; l--)
      for (int c=0; c<f.cols; c++)
        f(l,c)=f(l-1,c);
    vec_t w; converte(f,w);
    return w;
  } else if (ns==1) {
    Mat_<FLT> f; converte(v,f,nl,nc);
    for (int l=f.rows-1; l>0; l--)
      for (int c=0; c<f.cols; c++)
        f(l,c)=f(l-1,c);
```

```
    vec_t w; converte(f,w);
    return w;
  } else xerro1("Erro deslocaBaixo: deve ter ns==1 ou ns==3");
}

vec_t espelha(const vec_t& v, int nl, int nc, int ns) {
  //ns = numero de slices ou numero de espectros ou numero de cores
  //Neste momento, ns==1 ou ns==3
  if (v.size()!=unsigned(nl*nc*ns)) xerro1("Erro: Dimensao invalida");
  if (ns==3) {
    Mat_<CORF> f; converte(v,f,nl,nc);
    flip(f,f,1);
    vec_t w; converte(f,w);
    return w;
  } else if (ns==1) {
    Mat_<FLT> f; converte(v,f,nl,nc);
    flip(f,f,1);
    vec_t w; converte(f,w);
    return w;
  } else xerro1("Erro espelha: deve ter ns==1 ou ns==3");
}

void aumentaTreino(vector<vec_t>& timg, vector<label_t>& tlab, int nl, int nc, int ns,
  bool bespelha=false, bool diagonais=false) {
  if (timg.size()!=tlab.size()) xerro1("Erro: Dimensoes diferentes timg e tlab");
  if (ns!=1 && ns!=3) xerro1("Erro aumentaTreino: deve ter ns==1 ou ns==3");

  if (bespelha) {
    unsigned tamanho=timg.size();
    for (unsigned i=0; i<tamanho; i++) {
      timg.push_back( espelha( timg[i],nl,nc,ns ) ); tlab.push_back( tlab[i] );
    }
  }

  unsigned tamanho=timg.size();
  for (unsigned i=0; i<tamanho; i++) {
    timg.push_back( deslocaEsquerda( timg[i],nl,nc,ns ) ); tlab.push_back( tlab[i] );
    timg.push_back( deslocaDireita(  timg[i],nl,nc,ns ) ); tlab.push_back( tlab[i] );
    timg.push_back( deslocaCima( timg[i],nl,nc,ns ) );     tlab.push_back( tlab[i] );
    timg.push_back( deslocaBaixo(  timg[i],nl,nc,ns ) );   tlab.push_back( tlab[i] );
  }

  if (diagonais) {
    unsigned tamanho=timg.size();
    for (unsigned i=0; i<tamanho; i++) {
      timg.push_back( deslocaCima( deslocaEsquerda( timg[i],nl,nc,ns ), nl,nc,ns) );
      tlab.push_back( tlab[i] );
      timg.push_back( deslocaCima( deslocaDireita(  timg[i],nl,nc,ns ), nl,nc,ns) );
      tlab.push_back( tlab[i] );
      timg.push_back( deslocaBaixo( deslocaEsquerda( timg[i],nl,nc,ns ), nl,nc,ns) );
      tlab.push_back( tlab[i] );
      timg.push_back( deslocaBaixo( deslocaDireita(  timg[i],nl,nc,ns ), nl,nc,ns) );
      tlab.push_back( tlab[i] );
    }
  }
}
```

Nota 1: Em todas as conversões, estou supondo que os pixels do tiny_dnn são do tipo float que vai de 0 a 1 (preto a branco).

Nota 2: vec_t é o mesmo que vector<float>, mas com alinhamento de memória para poder usar eficientemente as instruções SIMD (AVX, SSE, etc). O tipo vec_t não armazena nenhuma informação sobre linha/coluna.

Nota 3: Tiny_dnn armazena uma imagem em níveis de cinza em vec_t utilizando a ordem raster (linha-coluna), por exemplo: [v(0,0), v(0,1), v(1,0), v(1,1)].

Nota 4: Tiny_dnn armazena uma imagem colorida em vec_t utilizando a ordem canal_RGB-linha-coluna. Isto é diferente da convenção de OpenCV (linha-coluna-canal_BGR). Por exemplo, no tiny_dnn: [R(0,0), R(0,1), R(1,0), R(1,1), G(0,0), G(0,1), G(1,0), G(1,1), B(0,0), B(0,1), B(1,0), B(1,1)]. No OpenCV: [B(0,0), G(0,0), R(0,0), B(0,1), G(0,1), R(0,1), B(1,0), G(1,0), R(1,0), B(1,1), G(1,1), R(1,1)]. Veja a seção "reading images" do manual da tiny_dnn.

```
//cnn10.cpp
//Este programa usa tiny_dnn e Cekeikon.
//compila cnn10 -t -c
//Uma nova versao de aumentaTreino2 esta dentro do cektiny.h
#include <cektiny.h>

string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main() {
  int n_train_epochs=30;
  int n_minibatch=10;

  network<sequential> net;
  net << conv(28, 28, 5, 1, 20) << relu()
      << max_pool(24, 24, 20, 2) << relu()
      << conv(12, 12, 5, 20, 40) << relu()
      << max_pool(8, 8, 40, 2) << relu()
      << fc(4*4*40, 1000) << relu()
      << fc(1000, 10) << sigmoid();

  vector<label_t> train_labels;
  vector<vec_t> train_images;
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, 0.0, 1.0, 0, 0);
  aumentaTreino(train_images,train_labels,28,28,1);

  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images, 0.0, 1.0, 0, 0);

  adagrad optimizer;

  cout << "start training" << endl;
  progress_display disp(train_images.size());
  timer t;

  int epoch = 1;
  auto on_enumerate_epoch = [&]() {
    cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
              << t.elapsed() << "s elapsed." << endl;
    ++epoch;
    result res = net.test(test_images, test_labels);
    cout << res.num_success << "/" << res.num_total << endl;

    disp.restart(train_images.size());
    t.restart();
  };

  auto on_enumerate_minibatch = [&]() { disp += n_minibatch; };

  net.train<mse>(optimizer, train_images, train_labels, n_minibatch,
                 n_train_epochs, on_enumerate_minibatch, on_enumerate_epoch);

  cout << "end training." << endl;
}
```

```
Epoch1=99,07% (0,93% de erro)
Epoch2=99,26% (0,74% de erro)
Epoch3=99,39% (0,61% de erro)
Epoch4=99,41% (0,59% de erro)
Epoch5=99,43% (0,57% de erro)
Epoch7=99,45% (0,55% de erro)
Epoch8=99,44%
Demora 476s cada epoch.
```

Aqui, os testes foram feitos com ativação relu após max_pool. Se retirasse, talvez a taxa de erro fosse menor.

Estamos cometendo um erro teórico aqui. Estamos usando o conjunto de teste para verificar o momento em que o menor erro é atingido. Isto está errado. Deve-se utilizar conjunto de validação para detectar a rede com o menor erro de validação. Depois, deve-se aplicar essa rede no conjunto de teste para calcular o erro de teste.

Agora, vamos eliminar relu da saída de max_pool e usar a técnica dropout (veja o livro para maiores detalhes).

```
//cnn12.cpp
  network<sequential> net;
  net << conv(28, 28, 5, 1, 20) << relu()
      << max_pool(24, 24, 20, 2)
      << conv(12, 12, 5, 20, 40) << relu()
      << max_pool(8, 8, 40, 2)
      << fc(4*4*40, 1000) << relu() << dropout(1000,0.5)
      << fc(1000, 10) << sigmoid();
```

Epoch1=99,26%. Demora 470 s/epoch.
Epoch2=99,45%.
Epoch6=99,53%. (erro 0,47%)
Epoch7=99,52%.
Epoch13=99,52%

O livro relata que conseguiu acuracidade de 99.60% (erro 0.40%) usando uma única rede. Chegamos numa taxa de erro próxima (erro 0.47%).

O livro relata acuracidade de 99.67% (erro 0,37%) usando comitê de 5 redes neurais. Não fiz teste semelhante (demoraria muito).

Vamos pegar cnn_noact2.cpp e tentar melhorá-la ainda mais.

A descrição detalhada destes testes está na apostila mnist.odt (não disponibilizada publicamente).

```
//cnn_noact2.cpp
#include "tiny_dnn/tiny_dnn.h"
#include <string>
using namespace tiny_dnn;
using namespace tiny_dnn::activation;
using namespace tiny_dnn::layers;
using namespace std;
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main() {
  int n_train_epochs=30;
  int n_minibatch=10;

  network<sequential> net;
  net << conv(28, 28, 5, 1, 20)
      << max_pool(24, 24, 20, 2) << relu()
      << conv(12, 12, 5, 20, 40)
      << max_pool(8, 8, 40, 2) << relu()
      << fc(4*4*40, 1000) << relu()
      << fc(1000, 10) << sigmoid();

  vector<label_t> train_labels;
  vector<vec_t> train_images;
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, 0.0, 1.0, 0, 0);
  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images, 0.0, 1.0, 0, 0);

  adagrad optimizer;

  cout << "start training" << endl;
  progress_display disp(train_images.size());
  timer t;

  int epoch = 1;
  int omelhor=-1;
  auto on_enumerate_epoch = [&]() {
    cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
              << t.elapsed() << "s elapsed.";
    ++epoch;
    result res = net.test(test_images, test_labels);
    cout << res.num_success << "/" << res.num_total << endl;

    if (omelhor<res.num_success) {
      omelhor=res.num_success;
      string nomearq = semDiret(string(argv[0]))+".net";
      net.save(nomearq);
      cout << "Gravado arquivo " << nomearq << endl;
    }

    disp.restart(train_images.size());
    t.restart();
  };

  auto on_enumerate_minibatch = [&]() { disp += n_minibatch; };

  net.train<mse>(optimizer, train_images, train_labels, n_minibatch,
                 n_train_epochs, on_enumerate_minibatch, on_enumerate_epoch);

  cout << "end training." << endl;
}
```

```
Epoch1=98,69%.  Demora 150s por epoch sem TBB. Demora 75s por epoch usando TBB.
Epoch2=98,96%
Epoch7=99,23%.
Epoch11=99,37%.
Epoch13=99,41% (erro=0,59%).
Epoch14=99,39%
Epoch30=99,25%
```

Eliminando o último sigmoide (ficou pouquinho melhor):

```
Epoch 1/30 finished. 73.0519s elapsed. 9870/10000
Epoch 2/30 finished. 72.9673s elapsed. 9895/10000
Epoch 3/30 finished. 72.5879s elapsed. 9909/10000
Epoch 4/30 finished. 73.1878s elapsed. 9915/10000
Epoch 5/30 finished. 80.2419s elapsed. 9917/10000
Epoch 6/30 finished. 82.5229s elapsed. 9926/10000
Epoch 7/30 finished. 81.9782s elapsed. 9926/10000
```

Vamos usar otimizador Adam:

cnn_adam10.cpp:
optimizer.alpha *= 0.1;
n_minibatch=10;
Dado de treino aumentado 4x.

```
//cnn_adam10.cpp
#include <cektiny.h>
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main(int argc, char **argv) {
  int n_train_epochs=30;
  int n_minibatch=10;

  network<sequential> net;
  net << conv(28, 28, 5, 1, 20)
      << max_pool(24, 24, 20, 2) << relu()
      << conv(12, 12, 5, 20, 40)
      << max_pool(8, 8, 40, 2) << relu()
      << fc(4*4*40, 1000) << relu()
      << fc(1000, 10);

  vector<label_t> train_labels;
  vector<vec_t> train_images;
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, 0.0, 1.0, 0, 0);
  aumentaTreino(train_images,train_labels,28,28,1,false);

  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images, 0.0, 1.0, 0, 0);

  adam optimizer;
  optimizer.alpha *= 0.1;

  cout << "start training" << endl;
  progress_display disp(train_images.size());
  timer t;

  int epoch = 1;
  int omelhor=-1;
  auto on_enumerate_epoch = [&]() {
    cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
              << t.elapsed() << "s elapsed. ";
    ++epoch;
    result res = net.test(test_images, test_labels);
    cout << res.num_success << "/" << res.num_total << endl;

    if (omelhor<res.num_success) {
      omelhor=res.num_success;
      string nomearq = semDiret(string(argv[0]))+".net";
      net.save(nomearq);
      cout << "Gravado arquivo " << nomearq << endl;
    }

    disp.restart(train_images.size());
    t.restart();
  };
```

Parece que learning rate inicial é 0.0001

```
Epoch 1/30 finished. 652.36s elapsed. 9923/10000
Epoch 2/30 finished. 650.614s elapsed. 9941/10000
Epoch 3/30 finished. 649.498s elapsed. 9943/10000
Epoch 4/30 finished. 645.865s elapsed. 9947/10000
Epoch 5/30 finished. 647.25s elapsed. 9951/10000
Epoch 6/30 finished. 649.203s elapsed. 9954/10000
Epoch 7/30 finished. 636.135s elapsed. 9955/10000
Epoch 8/30 finished. 377.925s elapsed. 9959/10000
Epoch 9/30 finished. 378.181s elapsed. 9959/10000
Epoch 10/30 finished. 380.442s elapsed. 9958/10000
Epoch 11/30 finished. 380.088s elapsed. 9959/10000
Epoch 12/30 finished. 380.423s elapsed. 9959/10000
Epoch 13/30 finished. 381.385s elapsed. 9958/10000
Epoch 14/30 finished. 381.021s elapsed. 9959/10000
Epoch 15/30 finished. 382.677s elapsed. 9959/10000
Epoch 16/30 finished. 382.529s elapsed. 9958/10000
Epoch 17/30 finished. 383.744s elapsed. 9958/10000
Epoch 18/30 finished. 384.395s elapsed. 9959/10000
Epoch 19/30 finished. 384.448s elapsed. 9958/10000
Epoch 20/30 finished. 385.852s elapsed. 9957/10000
Epoch 21/30 finished. 386s elapsed. 9957/10000
Epoch 22/30 finished. 386.81s elapsed. 9956/10000
Epoch 23/30 finished. 387.967s elapsed. 9955/10000
Epoch 24/30 finished. 389.625s elapsed. 9954/10000
Epoch 25/30 finished. 390.114s elapsed. 9955/10000
Epoch 26/30 finished. 389.843s elapsed. 9954/10000
Epoch 27/30 finished. 391.27s elapsed. 9955/10000
Epoch 28/30 finished. 392.723s elapsed. 9954/10000
Epoch 29/30 finished. 392.049s elapsed. 9955/10000
Epoch 30/30 finished. 393.505s elapsed. 9956/10000
```

cnn_adam18:
Com 3 camadas fully-connected.
Sem aumento de dados de treinamento.
Note que a leitura agora é feita -1.0 (preto) a +1.0 (branco)

```
  network<sequential> net;
  net << conv(28, 28, 5, 1, 20)
      << max_pool(24, 24, 20, 2) << relu()
      << conv(12, 12, 5, 20, 40)
      << max_pool(8, 8, 40, 2) << relu()
      << fc(4*4*40, 1000) << relu()
      << fc(1000, 100) << relu()
      << fc(100, 10);
  ...
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, -1.0, 1.0, 0, 0);
  //aumentaTreino(train_images,train_labels,28,28,1,false);
  ...
  adam optimizer;
  optimizer.alpha *= 0.1;
```

```
Epoch 1/30 finished. 107.092s elapsed. 9881/10000
Epoch 2/30 finished. 108.524s elapsed. 9923/10000
Epoch 3/30 finished. 108.814s elapsed. 9928/10000
Epoch 4/30 finished. 109.435s elapsed. 9939/10000
Epoch 5/30 finished. 108.604s elapsed. 9942/10000
Epoch 6/30 finished. 108.953s elapsed. 9950/10000
Epoch 7/30 finished. 107.972s elapsed. 9949/10000
Epoch 8/30 finished. 107.948s elapsed. 9948/10000
Epoch 9/30 finished. 107.89s elapsed. 9951/10000
Epoch 10/30 finished. 107.191s elapsed. 9953/10000
Epoch 11/30 finished. 107.945s elapsed. 9951/10000
Epoch 12/30 finished. 108.376s elapsed. 9954/10000
Epoch 13/30 finished. 108.699s elapsed. 9954/10000
Epoch 14/30 finished. 108.688s elapsed. 9954/10000
Epoch 15/30 finished. 108.688s elapsed. 9953/10000
Epoch 16/30 finished. 108.912s elapsed. 9953/10000
Epoch 17/30 finished. 108.819s elapsed. 9951/10000
Epoch 18/30 finished. 109.534s elapsed. 9951/10000
Epoch 19/30 finished. 109.507s elapsed. 9951/10000
Epoch 20/30 finished. 110.466s elapsed. 9953/10000
Epoch 21/30 finished. 109.907s elapsed. 9955/10000
Epoch 22/30 finished. 109.172s elapsed. 9953/10000
Epoch 23/30 finished. 108.693s elapsed. 9954/10000
Epoch 24/30 finished. 116.2s elapsed. 9955/10000
Epoch 25/30 finished. 111.46s elapsed. 9955/10000
Epoch 26/30 finished. 110.522s elapsed. 9957/10000
Epoch 27/30 finished. 109.089s elapsed. 9955/10000
Epoch 28/30 finished. 109.063s elapsed. 9956/10000
Epoch 29/30 finished. 109.246s elapsed. 9953/10000
Epoch 30/30 finished. 109.275s elapsed. 9955/10000
```

O melhor sem aumento de treino.
Chegou a 9957. No fim, chegou a 9955. Erro de 0,45%.

cnn_adam25: Igual a cnn_adam18 com aumento de dados de treinamento

```
  optimizer.alpha *= 0.01;

Epoch 1/30 finished. 1073.74s elapsed. 9818/10000
Epoch 2/30 finished. 1133.15s elapsed. 9880/10000
Epoch 3/30 finished. 1141.89s elapsed. 9910/10000
Epoch 4/30 finished. 1100.8s elapsed. 9922/10000
Epoch 5/30 finished. 1131.67s elapsed. 9935/10000
Epoch 6/30 finished. 1127.13s elapsed. 9941/10000
Epoch 7/30 finished. 1103.62s elapsed. 9943/10000
Epoch 8/30 finished. 1091.65s elapsed. 9944/10000
Epoch 9/30 finished. 1136.33s elapsed. 9946/10000
Epoch 10/30 finished. 1114.37s elapsed. 9949/10000
Epoch 11/30 finished. 1114.5s elapsed. 9950/10000
Epoch 12/30 finished. 1103.58s elapsed. 9951/10000
Epoch 13/30 finished. 1094.48s elapsed. 9952/10000
Epoch 14/30 finished. 1123.04s elapsed. 9951/10000
Epoch 15/30 finished. 1094.55s elapsed. 9953/10000
Epoch 16/30 finished. 1129.8s elapsed. 9951/10000
Epoch 17/30 finished. 1135.41s elapsed. 9951/10000
Epoch 18/30 finished. 1096.38s elapsed. 9953/10000
Epoch 19/30 finished. 1098.85s elapsed. 9956/10000
Epoch 20/30 finished. 1090.79s elapsed. 9955/10000
Epoch 21/30 finished. 1120.03s elapsed. 9956/10000
Epoch 22/30 finished. 1093.06s elapsed. 9956/10000
Epoch 23/30 finished. 1092.4s elapsed. 9957/10000
Epoch 24/30 finished. 1114.62s elapsed. 9958/10000
Epoch 25/30 finished. 1131.52s elapsed. 9958/10000
Epoch 26/30 finished. 1098.53s elapsed. 9958/10000
Epoch 27/30 finished. 1095.13s elapsed. 9958/10000
Epoch 28/30 finished. 1129.99s elapsed. 9959/10000
Epoch 29/30 finished. 1104.36s elapsed. 9959/10000
Epoch 30/30 finished. 1107.59s elapsed. 9960/10000
Gravado arquivo cnn_adam25.net
```

```
//adapt01.cpp: Igual ao cnn_adam25.cpp
//  Com inicial optimizer.alpha *= 0.03;
//  Multiplica alpha por 0.5 quando para de melhorar
// $compila adapt01 -c -t
#include <cektiny.h>
string data_dir_path="/home/hae/cekeikon5/tiny_dnn/data/";

int main(int argc, char **argv) {
  int n_train_epochs=30;
  int n_minibatch=10;

  network<sequential> net;
  net << conv(28, 28, 5, 1, 20)
      << max_pool(24, 24, 20, 2) << relu()
      << conv(12, 12, 5, 20, 40)
      << max_pool(8, 8, 40, 2) << relu()
      << fc(4*4*40, 1000) << relu()
      << fc(1000, 100) << relu()
      << fc(100, 10);

  vector<label_t> train_labels;
  vector<vec_t> train_images;
  parse_mnist_labels(data_dir_path + "train-labels.idx1-ubyte", &train_labels);
  parse_mnist_images(data_dir_path + "train-images.idx3-ubyte", &train_images, -1.0, 1.0, 0, 0);
  aumentaTreino(train_images,train_labels,28,28,1,false);

  vector<label_t> test_labels;
  vector<vec_t> test_images;
  parse_mnist_labels(data_dir_path + "t10k-labels.idx1-ubyte", &test_labels);
  parse_mnist_images(data_dir_path + "t10k-images.idx3-ubyte", &test_images, -1.0, 1.0, 0, 0);

  adam optimizer;
  optimizer.alpha *= 0.03;
  cout << "Learning rate=" << optimizer.alpha << endl;
  int sucesso_anterior=0;

  cout << "start training" << endl;
  progress_display disp(train_images.size());
  timer t;

  int epoch = 1;
  int omelhor=-1;
  auto on_enumerate_epoch = [&]() {
    cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
              << t.elapsed() << "s elapsed. ";
    ++epoch;
    result res = net.test(test_images, test_labels);
    cout << res.num_success << "/" << res.num_total << endl;
    if (res.num_success<=sucesso_anterior) {
      optimizer.alpha *= 0.5;
      cout << "Learning rate=" << optimizer.alpha << endl;
    }
    sucesso_anterior=res.num_success;

    if (omelhor<=res.num_success) {
      omelhor=res.num_success;
      string nomearq = semDiret(string(argv[0]))+".net";
      net.save(nomearq);
      cout << "Gravado arquivo " << nomearq << endl;
    }

    disp.restart(train_images.size());
    t.restart();
  };

  auto on_enumerate_minibatch = [&]() { disp += n_minibatch; };

  net.train<mse>(optimizer, train_images, train_labels, n_minibatch,
                 n_train_epochs, on_enumerate_minibatch, on_enumerate_epoch);

  cout << "end training." << endl;
}
```

```
Learning rate=3e-05
Epoch 1/30 finished. 1060.59s elapsed. 9899/10000
Epoch 2/30 finished. 1052.46s elapsed. 9936/10000
Epoch 3/30 finished. 1061.39s elapsed. 9946/10000
Epoch 4/30 finished. 1075.37s elapsed. 9952/10000
Epoch 5/30 finished. 1076.05s elapsed. 9954/10000
Epoch 6/30 finished. 1079.97s elapsed. 9957/10000
Epoch 7/30 finished. 1081.5s elapsed. 9958/10000
Epoch 8/30 finished. 1081.48s elapsed. 9958/10000
Learning rate=1.5e-05
Epoch 9/30 finished. 1082.73s elapsed. 9962/10000
Epoch 10/30 finished. 1082.79s elapsed. 9963/10000
Epoch 11/30 finished. 1085.43s elapsed. 9962/10000
Learning rate=7.5e-06
Epoch 12/30 finished. 1086.19s elapsed. 9964/10000
Epoch 13/30 finished. 1088.2s elapsed. 9963/10000
Learning rate=3.75e-06
Epoch 14/30 finished. 1085.83s elapsed. 9966/10000
Gravado arquivo adapt01.net
Epoch 15/30 finished. 1086.83s elapsed. 9966/10000 (0.34%)
Learning rate=1.875e-06
Epoch 16/30 finished. 1089.26s elapsed. 9964/10000
Learning rate=9.375e-07
Epoch 17/30 finished. 1086.02s elapsed. 9963/10000
Learning rate=4.6875e-07
Epoch 18/30 finished. 1085.28s elapsed. 9962/10000
Learning rate=2.34375e-07
Epoch 19/30 finished. 1084.02s elapsed. 9961/10000
Learning rate=1.17188e-07
Epoch 20/30 finished. 1089.97s elapsed. 9961/10000
Learning rate=5.85938e-08
Epoch 21/30 finished. 1089.53s elapsed. 9961/10000
Learning rate=2.92969e-08
Epoch 22/30 finished. 1088.2s elapsed. 9961/10000
Learning rate=1.46484e-08
Epoch 23/30 finished. 1087.18s elapsed. 9961/10000
Learning rate=7.32422e-09
Epoch 24/30 finished. 1088.44s elapsed. 9961/10000
Learning rate=3.66211e-09
Epoch 25/30 finished. 1084.81s elapsed. 9961/10000
Learning rate=1.83105e-09
Epoch 26/30 finished. 1086.7s elapsed. 9961/10000
Learning rate=9.15527e-10
Epoch 27/30 finished. 1072.55s elapsed. 9961/10000
Learning rate=4.57764e-10
Epoch 28/30 finished. 1081.79s elapsed. 9961/10000
Learning rate=2.28882e-10
Epoch 29/30 finished. 1078.68s elapsed. 9961/10000
Learning rate=1.14441e-10
Epoch 30/30 finished. 1081.14s elapsed. 9961/10000
Learning rate=5.72205e-11
```

O resultado é comparável ao estado da arte:
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

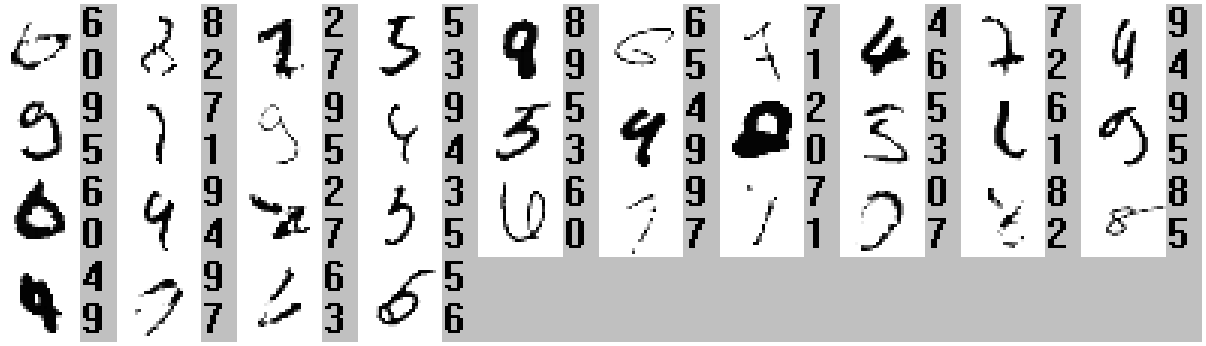
//adapt02.cpp: Igual ao cnn_adam25.cpp
// Com inicial optimizer.alpha *= 0.03;
// Multiplica alpha por 0.5 quando para de melhorar
// Nao deixa optimizer.alpha ser menor que 4e-6

```
    if (res.num_success<=sucesso_anterior && optimizer.alpha>4e-6) {
      optimizer.alpha *= 0.5;
      cout << "Learning rate=" << optimizer.alpha << endl;
    }
```

```
Learning rate=3e-05
Epoch 1/30 finished. 952.539s elapsed. 9899/10000
Epoch 2/30 finished. 1047.25s elapsed. 9936/10000
Epoch 3/30 finished. 1058.48s elapsed. 9946/10000
Epoch 4/30 finished. 1063.38s elapsed. 9952/10000
Epoch 5/30 finished. 1076s elapsed. 9954/10000
Epoch 6/30 finished. 1110.25s elapsed. 9957/10000
Epoch 7/30 finished. 1097.49s elapsed. 9958/10000
Epoch 8/30 finished. 1094.76s elapsed. 9958/10000
Learning rate=1.5e-05
Epoch 9/30 finished. 1096.2s elapsed. 9962/10000
Epoch 10/30 finished. 1091.47s elapsed. 9963/10000
Epoch 11/30 finished. 1095.59s elapsed. 9962/10000
Learning rate=7.5e-06
Epoch 12/30 finished. 1091.82s elapsed. 9964/10000
Epoch 13/30 finished. 1110.16s elapsed. 9963/10000
Learning rate=3.75e-06
Epoch 14/30 finished. 1091.36s elapsed. 9966/10000
Gravado arquivo adapt02.net
Epoch 15/30 finished. 1116.49s elapsed. 9966/10000
Gravado arquivo adapt02.net
Epoch 16/30 finished. 1109.54s elapsed. 9965/10000
Epoch 17/30 finished. 1103.43s elapsed. 9966/10000
Gravado arquivo adapt02.net
Epoch 18/30 finished. 1107.11s elapsed. 9965/10000
Epoch 19/30 finished. 1109.77s elapsed. 9966/10000
Gravado arquivo adapt02.net
Epoch 20/30 finished. 1137.92s elapsed. 9965/10000
Epoch 21/30 finished. 1163.67s elapsed. 9965/10000
Epoch 22/30 finished. 1152.97s elapsed. 9965/10000
Epoch 23/30 finished. 1170.94s elapsed. 9965/10000
Epoch 24/30 finished. 1160.2s elapsed. 9964/10000
Epoch 25/30 finished. 1139.47s elapsed. 9963/10000
Epoch 26/30 finished. 1158.01s elapsed. 9964/10000
Epoch 27/30 finished. 1142.11s elapsed. 9964/10000
Epoch 28/30 finished. 1133.79s elapsed. 9964/10000
Epoch 29/30 finished. 1175.66s elapsed. 9964/10000
Epoch 30/30 finished. 1148.36s elapsed. 9963/10000
```

O melhor com aumento de treino? O melhor 9966. No fim, chegou a 9963.

Conclusão: Conseguimos erro de 0,34% de validação ou 0,37% de teste. É melhor que a taxa de erro relatada no livro (erro 0.40%) usando uma única rede. O resultado ainda é pior que o estado da arte 0,21%.
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

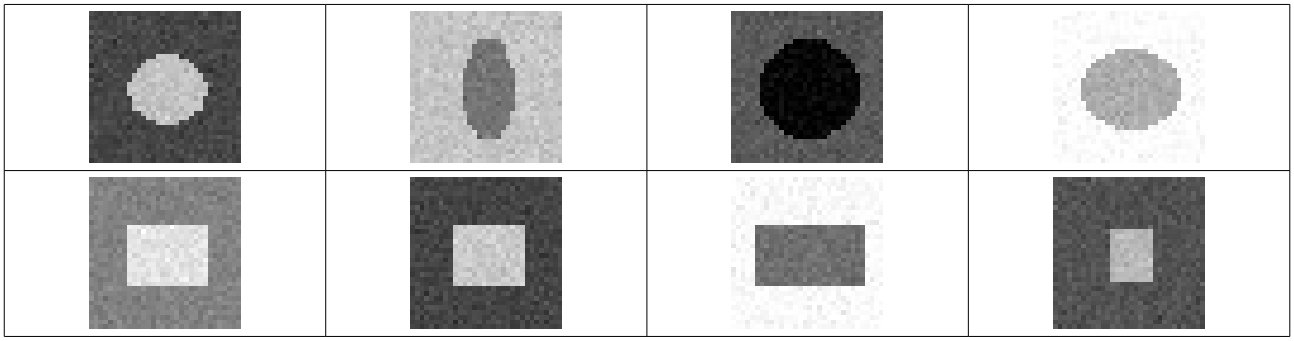


34 caracteres classificados incorretamente. Superior-direita: classificação verdadeira; inferior-direita: classificação atribuída pelo algoritmo. Um ser humano também teria dificuldade para classificar a maioria dessas imagens.

Distinguir elipse e retângulo.

Vamos mostrar um exemplo onde treino/teste é feito usando imagens comuns (e não a partir de um banco de dados num formato especial). Também relembrar como é possível salvar/carregar a rede treinada.

Descompactando o arquivo eliret.zip, obtemos 200 imagens de elipses e 200 imagens de retângulos de diferentes tamanhos, excentricidades, níveis de cinza e com (pequeno) ruído gaussiano. Todas as imagens são níveis de cinza com 32x32 pixels. Essas imagens podem ser geradas rodando o programa gera.cpp (que está dentro do eliret.zip).



O objetivo é, dada uma imagem, reconhecer se é elipse ou retângulo. As 400 imagens foram divididas em três conjuntos disjuntos:

- 200 imagens para treino (imagens de 000 a 099)
- 100 imagens para validação (imagens de 100 a 149)
- 100 imagens para teste (imagens de 150 a 199)

Essas divisões estão escritas em 3 arquivos: treino.csv, valida.csv e teste.csv. Após o nome da imagem, aparece o rótulo (0=elipse, 1=retângulo)

Arquivo treino.csv:

```
eli000.png;0
eli001.png;0
...
eli099.png;0
ret000.png;1
ret001.png;1
...
ret099.png;1
```

Arquivo valida.csv:

```
eli100.png;0
...
eli149.png;0
ret100.png;1
...
ret149.png;1
```

Arquivo teste.csv:

```
eli150.png;0
...
eli199.png;0
ret150.png;1
...
ret199.png;1
```

Rodando treina.cpp, temos:

```
$crono treina
Epoch 1/10 finished. 8.4616s elapsed. Validacao: 50/100
Epoch 2/10 finished. 8.88227s elapsed. Validacao: 50/100
Epoch 3/10 finished. 9.04181s elapsed. Validacao: 50/100
Epoch 4/10 finished. 8.88111s elapsed. Validacao: 50/100
Epoch 5/10 finished. 8.94792s elapsed. Validacao: 84/100
Epoch 6/10 finished. 8.935s elapsed. Validacao: 96/100
Epoch 7/10 finished. 8.97272s elapsed. Validacao: 98/100
Epoch 8/10 finished. 8.93749s elapsed. Validacao: 99/100
Epoch 9/10 finished. 8.91424s elapsed. Validacao: 100/100
Epoch 10/10 finished. 8.93154s elapsed. Validacao: 100/100
Tempo gasto      1.503 minutos
```

Rodando testa.cpp, temos:

```
$crono testa
<<<<< validacao <<<<<<
accuracy:100% (100/100)
    *     0     1
    0    50     0
    1     0    50
<<<<< teste <<<<<<
accuracy:100% (100/100)
    *     0     1
    0    50     0
    1     0    50
Tempo gasto      0.478 segundos
```

Isto é, acerto de 100% (tanto de validação como de teste).

Copio abaixo os programas usadas neste problema:

```
//gera.cpp
//gera elipses e retangulos
#include <cekeikon.h>

int main() {
  RNG rng(7);
  int n=200;

  for (int i=0; i<n; i++) {
    GRY corfundo=rng(256);
    GRY corelipse=rng(256);
    while (abs(corfundo-corelipse)<50)
      corelipse=rng(256);
    Mat_<GRY> a(32,32,corfundo);
    int tamx=a.cols/8+rng(a.cols/4);
    int tamy=a.rows/8+rng(a.rows/4);
    ellipse(a,Point(a.cols/2,a.rows/2),Size(tamx,tamy),
            0,0,360,toScalar(corelipse),-1);
    for (unsigned i=0; i<a.total(); i++)
      a(i) = saturate_cast<GRY>(a(i) + rng.gaussian(8));
    imp(a,format("eli%03d.png",i));
  }

  for (int i=0; i<n; i++) {
    GRY corfundo=rng(256);
    GRY corret=rng(256);
    while (abs(corfundo-corret)<50)
      corret=rng(256);
    Mat_<GRY> a(32,32,corfundo);
    int tamx=a.cols/8+rng(a.cols/4);
    int tamy=a.rows/8+rng(a.rows/4);
    int centrox=a.cols/2;
    int centroy=a.rows/2;
    rectangle(a,Point(centrox-tamx,centroy-tamy),Point(centrox+tamx,centroy+tamy),
              toScalar(corret),-1);
    for (unsigned i=0; i<a.total(); i++)
      a(i) = saturate_cast<GRY>(a(i) + rng.gaussian(8));
    imp(a,format("ret%03d.png",i));
  }
}
```

O programa abaixo faz leitura de imagens e rótulos através do arquivo CSV. Uso duas funções de Cekeikon para leitura de nome de arquivo e de número:

```
    string st=leFileName(arq);
    if (st=="eof") break;
    int rot; leNum(arq,rot);
```

Estas funções permitem escrever comentários estilo C++: /* ... */ e // ...

Talvez, se fizesse normalização para -1 a +1 ou -0.5 a +0.5 (em vez de 0 a 1) a convergência da rede seja mais rápida.

```
//csv.h - rotina para ler arquivo CSV com nomes de imagens e rotulos
//Sera usada pelos programas treina.cpp e testa.cpp
#include <cektiny.h>
#include <cekeikon.h>

//<<<<<<<<<<<<<<<<< IMAGEM_CSV <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
class ARQCSV {
//Le arquivo CSV com formato:
//111a.jpg;0
//111b.jpg;0
//Le imagem e label no mesmo diretorio do arquivo CSV e os coloca nos vetores x e y
 public:
  int n; // numero de linhas do CSV
  vector<vec_t> x; vector<label_t> y; // imagens e rotulos convertidos para formato tiny_dnn
  vector< string > nome;  // nomes dos arquivos
  ARQCSV(string caminho);
};

ARQCSV::ARQCSV(string arqcsv) {
  n=0;
  string nomedir=diretorio(arqcsv);
  if (nomedir=="") nomedir=".";
  FILE* arq=fopen(arqcsv.c_str(),"rt");
  while (true) {
    string st=leFileName(arq);
    if (st=="eof") break;
    int rot; leNum(arq,rot);
    n++;
  }
  x.resize(n);
  y.resize(n);
  nome.resize(n);
  rewind(arq);
  int i=0;
  Mat_<FLT> f;
  while (true) {
    string st=leFileName(arq);
    if (st=="eof") break;
    st=nomedir+'/'+st;
    int rot; leNum(arq,rot);
    le(f,st);
    //mostra(f);
    //resize(f,f,Size(nc,nl),0,0,INTER_AREA);
    normalize(f,f,1,0,NORM_MINMAX);
    //Provavelmente, e' melhor normalizar de -0.5 a +0.5 ou -1.0 a +1.0
    //mostra(f);
    converte(f,x[i]);
    y[i]=rot;
    nome[i]=st;
    i++;
  }
  fclose(arq);
}
```

```
//treina.cpp - treina rede para distinguir elipses e retangulos
//compila treina -c -t
#include "csv.h"

int main(int argc, char** argv) {
  ARQCSV a("treino.csv");
  ARQCSV v("valida.csv");

  int n_train_epochs=10;
  int n_minibatch=10;

  network<sequential> net;
  net << conv(32, 32, 5, 1, 40)
      << max_pool(28, 28, 40, 2) << relu()
      << conv(14, 14, 5, 40, 80)
      << max_pool(10, 10, 80, 2) << relu()
      << fc(5*5*80, 200) << relu()
      << fc(200, 20) << relu()
      << fc(20, 2);

  aumentaTreino(a.x,a.y,32,32,1,false,true); //espelha, diagonais

  adam optimizer;
  optimizer.alpha = 5e-5;
  cout << "Learning rate=" << optimizer.alpha << endl;
  int sucesso_anterior=0;

  cout << "start training" << endl;
  progress_display disp(a.x.size());
  timer t;

  int epoch = 1;
  int omelhor=-1;
  auto on_enumerate_epoch = [&]() {
    cout << "Epoch " << epoch << "/" << n_train_epochs << " finished. "
              << t.elapsed() << "s elapsed. ";
    ++epoch;
    result res = net.test(v.x, v.y);
    cout << "Validacao: " << res.num_success << "/" << res.num_total << endl;
    if (res.num_success<=sucesso_anterior && optimizer.alpha>5e-6) {
      optimizer.alpha *= 0.80;
      cout << "Learning rate=" << optimizer.alpha << endl;
    }
    sucesso_anterior=res.num_success;

    if (omelhor<=res.num_success) {
      omelhor=res.num_success;
      string nomearq = semDiret(string(argv[0]))+".net";
      net.save(nomearq);
      cout << "Gravado arquivo " << nomearq << endl;
    }

    disp.restart(a.x.size());
    t.restart();
  };

  auto on_enumerate_minibatch = [&]() { disp += n_minibatch; };

  net.train<mse>(optimizer, a.x, a.y, n_minibatch,
                 n_train_epochs, on_enumerate_minibatch, on_enumerate_epoch);

  cout << "end training." << endl;
}
```

```
//testa.cpp
//compila testa -c -t
//Taxa de erro de validacao: 0%
//Taxa de erro de teste: 0%
//Tempo de execucao: 0,5 s
#include "csv.h"

int main(int argc, char** argv) {
  ARQCSV v("valida.csv");
  ARQCSV q("teste.csv");
  network<sequential> net;
  net.load("treina.net");

  cout << "<<<<< validacao <<<<<<\n";
  net.test(v.x, v.y).print_detail(cout);
  for (unsigned i=0; i<v.x.size(); i++)
    if (net.predict_label(v.x[i])!=v.y[i])
      cout << v.nome[i] << endl;

  cout << "<<<<< teste <<<<<<\n";
  net.test(q.x, q.y).print_detail(cout);
  for (unsigned i=0; i<q.x.size(); i++)
    if (net.predict_label(q.x[i])!=q.y[i])
      cout << q.nome[i] << endl;
}
```

Aula 7, exercício 1 (2 pontos): Execute o programa treina.cpp obtendo treina.net. Execute o programa testa.cpp.

Aula 7, exercício 2 (2 pontos): Escreva o programa predicao.cpp que recebe o modelo treinado treina.net e um arquivo de elipse ou retângulo e responde se a imagem é elipse ou retângulo.

```
>predicao treina.net eli000.png
A imagem e' elipse.
>predicao treina.net ret000.png
A imagem e' retangulo.
```