Os computadores podem realizar muitas operações muito mais rápidas do que os humanos. **However, there are many tasks in which the computer falls considerably short**. Uma dessas tarefas é a interpretação de informação gráfica. Uma criança pré-escolar pode facilmente dizer a diferença entre um gato e um cachorro, mas esse problema simples confunde os computadores de hoje.

No campo da ciência da computação, inteligência artificial tenta dar habilidades humanas aos computadores. Um dos principais meios pelos quais os computadores são dotados com capacidades semelhantes às humanas é através da utilização de uma rede neural, onde o cérebro humano é o melhor exemplo disso. O cérebro humano consiste de uma rede de mais de um bilhão de neurônios interligados. Estas são células individuais que podem processar pequenas quantidades de informação e, em seguida, ativam outros neurônios para continuar o processo. No entanto, o termo rede neural, como é normalmente utilizado, é realmente um equívoco. Computadores tenta simular uma rede neural. A maioria das publicações usam o termo rede neural ao invés de rede neural artificial.

Este artigo mostra como construir uma rede neural em Java; No entanto, eles podem ser construídos em quase qualquer linguagem de programação. A maioria das publicações sobre redes neurais usam linguagens de computador como C, C ++, Lisp, ou Prolog. Java é atualmente uma linguagem de programação de rede neural bastante eficaz. Este artigo mostra um simples, yet particle, de como uma rede neural pode reconhecer letras manuscritas, e descreve a implementação de uma rede neural em um pequeno programa de exemplo.

**Reconhecendo Letras**

Usando o programa de exemplo (mostrado na Figura 1) você pode ver uma rede neural em ação. Para facilitar a distribuição, os arquivos de classe são empacotados em um único arquivo JAR nomeado OCR.jar. Para executar o programa, use o seguinte comando (supondo que você está no mesmo diretório que o arquivo JAR). Alguns sistemas podem permitir que você simplesmente clique duas vezes no arquivo JAR.

Quando o programa de reconhecimento de letra começa, não há dados carregados inicialmente. Um arquivo de treinamento deve ser utilizado que contém as formas das letras.  Um exemplo de arquivo de treinamento (**sample.dat**) é pré-carregado com as 26 letras maiúsculas. Para ver o programa de funcionando, clique no botão "**Load**", que carrega o arquivo sample.dat. Agora padrões de 26 letras estão na memória e na rede que devem ser treinados. Clique no botão "Begin Training"; Agora, a rede está pronta para reconhecer caracteres. Desenhe qualquer letra maiúscula que preferir e clique em "Reconhecer"; o programa deve agora reconhecer a letra.

**Treinamento do Programa de Amostra**

Talvez a minha escrita é consideravelmente diferente da maioria das pessoas. E se você quiser treinar o programa especificamente para sua escrita? Para substituir a letra que já foi definida, é necessário selecionar e apagar essa a letra primeiro. Basta pressionar o botão “**Delete**”.  Agora, desenhe o personagem que você deseja treinar o programa.  Se você preferir ver esta a uma amostra da letra abaixo antes de adicioná-lo, clique no botão "**downsample**". Se você está satisfeito com sua letra, clique no botão "**Add**" para adicioná-lo ao conjunto de treinamento. Para salvar uma cópia de suas letras recém-criadas, clique no botão "**Save**" que ele vai gravar no arquivo sample.dat.

Depois de inserir todas as letras que você quer, agora você deve “**train**” a rede neural.  Até este ponto você simplesmente forneceu um conjunto de treino de letras conhecidas como padrões de entrada. Com estes padrões de entrada, agora você está agora pronto para treinar a rede, o que poderia levar bastante tempo. No entanto, uma vez que apenas uma amostra de desenho por letra é permitido, este processo será concluído em questão de segundos. Uma pequena janela será exibida quando o treinamento estiver completo. Quando você salvar, apenas os padrões de caracteres são salvos. Se você carregar esses mesmos padrões mais tarde, você deve treinar novamente a rede.

Agora você irá mostrar como este programa de exemplo é construído, e como você pode criar programas similares. O arquivo **MainEntry.java** contém a aplicação **Swing** que compõe esta aplicação, que é pouco mais do que colocar os componentes em seus locais corretos.

As três áreas focadas neste artigo são **downsampling** (redução da resolução), **training** (treinamento) e **recognition** (reconhecimento). **Downsampling** (Diminuição da resolução), um algoritmo usado para reduzir a resolução das letras que estão sendo desenhadas, é usado para reconhecimento de caracteres e reconhecimento e treinamento, por isso vamos abordar este assunto primeiramente.

**Downsampling the Image**

Todas as imagens tem que ser **downsampled** antes de serem usadas, o que impede que a rede neural de se confundir pelo tamanho e posição. A área de desenho é grande o suficiente para que você possa desenhar uma letra em vários tamanhos diferentes. Por **downsampling** (diminuição da resolução) da imagem para um tamanho consistente, não importa o quão grande você desenhar a letra, como a imagem **downsampled** permanecerá sempre um tamanho consistente. Esta seção mostra como isso é feito.

Quando você desenhar uma imagem, o programa baseia-se em primeiramente na caixa em torno do limite de sua letra. Isso permite que o programa elimine todo o espaço em branco em torno de sua letra. Este processo é realizado no interior do método "**downsample**" da classe **Entry.java**. Como você desenhar um caracter, esse caracter também é desenhado para o "**entryImage**" variável de instância do objeto **Entry**. Para cortar esta imagem e, eventualmente, **downsample** **it**, devemos pegar o padrão de bits da imagem. Isso é feito usando uma classe **PixelGrabber**:

**Class** Entry

**public void** downSample(){

**int** w = entryImage.getWidth(**this**);

**int** h = entryImage.getHeight(**this**);

PixelGrabber grabber = **new** PixelGrabber(entryImage, 0, 0, w, h, **true**);

**try** {

grabber.grabPixels();

pixelMap = (**int**[]) grabber.getPixels(); [...]

Após este código ter terminado, a variável **pixelmap**, que é uma matriz de tipos de dados int, agora contém o padrão de bits da imagem. O próximo passo é cortar a imagem e remover qualquer espaço em branco. O corte é implementado arrastando quatro linhas imaginárias a partir do topo, à esquerda, baixo e lados direito da imagem. **These lines will stop as soon as they cross an actual pixel.** Ao fazer isto,  **these lines snap to the outer edges of the image.** Os métodos **hLineClear** e **vLineClear** aceitam ambos um parâmetro que indica qual linha examinar, e retorna **true** se essa linha **if that line is clear**. O programa funciona chamando **hLineClear** e **vLineClear** até eles cruzam as bordas externas da imagem. O método de linha horizontal (**hLineClear**) é mostrado aqui.

**Class** Entry

**protected** **boolean** hLineClear(**int** y) {

**int** w = entryImage.getWidth(**this**);

**for** (**int** i = 0; i < w; i++) {

**if** (pixelMap[(y \* w) + i] != -1)

**return** **false**;

}

**return** **true**;

}

Como você pode ver, o método da linha horizontal aceita uma coordenada y que verifica especificamente a linha horizontal. O programa, em seguida, percorre cada coordenada x nessa linha, verificando se há quaisquer valores de pixel. O valor de -1 indica branco, por isso é ignorado. O método "**findBounds**" usa "**hLineClear**" e "**vLineClear**" para calcular as quatro bordas. O início desse método é mostrado aqui:

**Class** Entry

**protected** **void** findBounds(**int** w, **int** h) {

// top line

**for** (**int** y = 0; y < h; y++) {

**if** (!hLineClear(y)) {

downSampleTop = y;

**break**;

}

}

// bottom line

**for** (**int** y = h - 1; y >= 0; y--) {

**if** (!hLineClear(y)) {

downSampleBottom = y;

**break**;

}

}

Você pode ver como o programa calcula as linhas superior e inferior do retângulo de corte. Para calcular a linha de topo, o programa começa em 0 e continua para a parte inferior da imagem. Assim que a primeira linha **non clear** for encontrado, o programa determina como o recorte superior do retângulo. O mesmo processo, mas ao contrário, é realizado para determinar a parte inferior da imagem. Os processos para determinar os limites de esquerdo e direito são realizadas da mesma maneira.

Agora que a imagem foi cortada, deve ser **downsampled**. Trata-se de pegar a imagem de uma maior resolução para uma resolução 5x7. Para reduzir uma imagem para 5x7, pense em uma grade imaginária sendo desenhada sobre a imagem de alta resolução. Isto divide a imagem em seções retangulares, cinco de largura e sete para altura. Se qualquer pixel numa seção estiver preenchido, o pixel correspondente na imagem de 5x7 **downsampled** também está preenchido.A maioria do trabalho realizado por este processo é realizado no interior do método "**downSampleQuadrant**" mostrado aqui.

**protected** **boolean** downSampleQuadrant(**int** x, **int** y) {

**int** w = entryImage.getWidth(**this**);

**int** startX = (**int**) (downSampleLeft + (x \* ratioX));

**int** startY = (**int**) (downSampleTop + (y \* ratioY));

**int** endX = (**int**) (startX + ratioX);

**int** endY = (**int**) (startY + ratioY);

**for** (**int** yy = startY; yy <= endY; yy++) {

**for** (**int** xx = startX; xx <= endX; xx++) {

**int** loc = xx + (yy \* w);

**if** (pixelMap[loc] != -1)

**return** **true**;

}

}

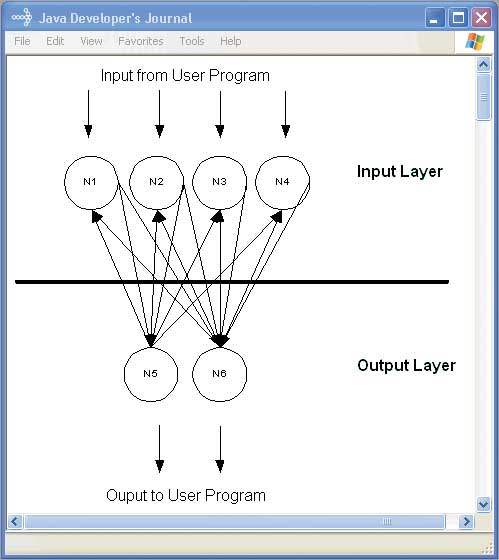
**return** **false**;

}

O método "**downSampleQuadrant**" aceita o número da seção que deve ser calculado. Primeiro o x inicial e final e as coordenadas y devem ser calculadas. Para calcular primeiro a coordenada x para a seção especificada, o primeiro "**downSampleLeft**" é usado; este é o lado esquerdo do retângulo de corte. Então x é multiplicado por "**ratioX**", a razão entre o número de pixels compõe cada seção. Isso nos permite determinar onde colocar "**startX**". A posição y inicial “**starty**”, é calculada por meios semelhantes. Em seguida, o programa percorre a cada x e y abrangidos pela seção especificada. Se até mesmo um pixel é determinada para ser preenchido, o método retorna **true**, o que indica que esta seção deve ser considerada preenchida.

O método "downSampleQuadrant" é chamado em sucessão para cada seção na imagem. Estes resultados de uma imagem de exemplo são armazenados na classe "**SampleData**", uma classe **wrapper** que contém uma matriz 5x7 de valores booleanos. É esta estrutura que forma a entrada para treinamento e reconhecimento de caracteres.

**Reconhecimento de Rede Neural**   
Há muitos tipos de redes neurais, e a maioria são nomeados após seus criadores. Eu usarei uma rede neural Kohonen, uma rede de dois níveis. O padrão de caractere **downsampled** desenhado pelo usuário é alimentado para os neurônios de entrada. Há um neurônio de entrada para cada pixel da imagem **downsampled**. Porque a imagem **downsampled** é uma grade 5x7, há 35 neurônios de entrada.



Através dos neurônios de saída, a rede neural comunica a cada letra que o usuário pensou e desenhou. O número de neurónios de saída sempre corresponde ao apenas ao tanto de letras foram fornecidas. Visto que foram fornecidas 26 letras na amostra, haverá 26 neurônios de saída. Se este programa for modificado para suportar múltiplas amostras de letras, ainda haveria 26 neurônios de saída, mesmo que houvesse múltiplas amostras por letra.

Em adição aos neurónios de entrada e saída, também existem ligações entre os neurónios individuais. Estas ligações não são todas iguais. Cada um é atribuído um peso, que em última análise é o único fator que determina qual a saída de rede para um determinado padrão de entrada. Para determinar o número total de conexões, multiplica-se o número de neurônios de entrada pelo número de neurônios de saída. Uma rede neural com 26 neurônios de saída e 35 neurônios de entrada terá um total de 910 pesos de conexões. O processo de **training** (treinamento) é dedicado a encontrar os valores corretos para esses pesos.

O processo de **recognition** (reconhecimento) começa quando o usuário desenha um caractere e, em seguida, clica no botão "**Recognize**". Primeira a letra será **downsampled** (reduzida) para uma imagem de 5x7. Esta imagem deve ser copiada a partir de sua matriz bidimensional para uma matriz de duplas que serão alimentados com os neurônios de entrada.

**Class** MainEntry

**void** recognize\_actionPerfomed(java.awt.event.ActionEvent event) {

entry.downSample();

**double** input[] = **new** **double**[5 \* 7];

**int** idx = 0;

SampleData ds = sample.getData();

**for** (**int** y = 0; y < ds.getHeight(); y++) {

**for** (**int** x = 0; x < ds.getWidth(); x++) {

input[idx++] = ds.getData(x, y) ? .5 : -.5;

}

}

[...]

Este código faz a conversão. Neurônios exigem entradas de ponto flutuante. Como resultado, o programa alimenta o valor de 5 para um pixel branco e -5 para um pixel preto. Esta array de 35 valores é alimentado para os neurónios de entrada, passando a matriz de entrada para o método "**winner** " do **Kohonen**. Isto devolve qual dos 35 neurónios ganhou e é armazenado no "melhor" inteiro

**Class** MainEntry

**void** recognize\_actionPerformed(Action Event event){

**int** best = net.winner(input, normfac, synth);

**char** map[] = mapNeurons();

JOptionPane.*showMessageDialog*(**this**, " " + map[best] + " (Neuron #" + best + " fired)", "That Letter Is",

JOptionPane.***PLAIN\_MESSAGE***);

clear\_actionPerformed(**null**);

[...]

Sabendo qual o neurônio vencedor não é muito útil, porque ele não mostra qual letra foi reconhecido. Para alinhar os neurônios com cada imagem da letra reconhecida, cada imagem de letra da rede foi treinada para ser alimentada dentro da rede e determinar o neurônio vencedor. Por exemplo, se você fosse alimentar a imagem de treinamento para "J" dentro da rede neural, e o neurônio vencedor for de numero #4, você sabe que ele é o único que tinha aprendido a reconhecer o padrão de J. Isso é feito chamando o método "**mapNeurons**", que retorna um array de caracteres. O índice de cada elemento da matriz corresponde ao número de neurónios que reconhece que o caractere.

A maior parte do trabalho real executado pela rede neural é feito no método **winner** . A primeira coisa que o método **winner** faz é normalizar as entradas e calcular os valores de saída para cada neurônio de saída. O neurônio de saída com o maior valor de saída é considerado o vencedor. Primeiro a " **biggest**" (maior) variável é definida para um número muito pequeno para indicar que não há vencedor ainda.

**Class** KohonenNetwork

**Public** **int** winner(double input[] ...){

biggest = -1.E30;

**for** (i = 0; i < outputNeuronCount; i++) {

optr = outputWeights[i];

output[i] = dotProduct(input, optr) \* normfac[0] + synth[0] \* optr[inputNeuronCount];

// Remap to bipolar(-1,1 to 0,1)

output[i] = 0.5 \* (output[i] + 1.0);

**if** (output[i] > biggest) {

biggest = output[i];

win = i;

}

Para cada peso de saída dos neurônios é calculado **taking the dot product** de cada peso dos neurônios de saída para os neurônios de entrada. **The** **dot product**  é calculado multiplicando cada um dos valores de entrada da entrada de neurónios contra os pesos entre esse neurónio de entrada e o neurónio de saída. Estes pesos foram determinados durante o treinamento, que é discutido na próxima seção. A saída é mantida, e se ele é o maior produto até o momento, ele é definido como o neurônio que venceu "**winnning**".

Como você pode ver obter os resultados de uma rede neural é um processo rápido. Na verdade determinar os pesos dos neurónios é a porção complexa desse processo. Treinamento da rede neural é discutido na seção seguinte.

**Como a Rede Neural Aprende**

Aprender é o processo de selecionar uma matriz de peso de neurônios que vai ser reconhecido corretamente na entrada padrão. Uma rede neural Kohonen aprende constantemente avaliando e aperfeiçoando uma matriz de peso. Para fazer isso, deve começar a partir de uma determinada matriz de pesos. Esta matriz é escolhida pra selecionar números aleatórios. Naturalmente, esta é uma escolha terrível para escolher uma matriz de pesos, mas dá um ponto de partida para aperfeiçoar a partir disso.

Uma vez que a matriz inicial de peso aleatório é criada, o treinamento pode começar. Primeiro, a matriz de peso é avaliada para determinar o qual é o seu atual nível de erro. Este erro é determinado pela forma como é feita o treinamento de entrada (as letras que você criou) mapeada para os neurônios de saída. O erro é calculado pelo método "evaluateErrors" da classe KohonenNetwork. Se o nível de erro é baixa, digamos, abaixo de 10%, o processo está completo.