APRENDIZADO DE MÁQUINA - TRABALHO 04

Redes Neurais

Jefferson Colares

Análise e conclusões sobre o código Python desenvolvido para este trabalho.

Professor:
Eduardo Bezerra

Rio de Janeiro,
Agosto, 2018

Sumário

Parte 1 - Redes Completamente Conectadas

Nessa parte do trabalho, é desenvolvida uma rede neural completamente conectada, utilizando a biblioteca scikit-learn, para inferir se um cliente irá ou não pargar um emprestimo contraído. O dataset utilizado para treinamento da rede contém 11 colunas de dados sobre empréstimos concedidos a 1500 clientes e uma coluna adicional informando se foram pagos ou não.

1.1 Localização dos arquivos

O código referente a essa parte do trabalho pode ser encontrado no arquivo **parte1/main.py**

1.2 Comentários sobre o código

Inicialmente, é executada a carga dos datasets de treino e teste em dataframes do Pandas:

```
# Carrega os conjuntos de dados de teste e treinamento
traindata = pd.read_table('credtrain.txt', header=None)
testdata = pd.read_table('credtest.txt', header=None)
```

Em seguida, cada dataset é separado em atributos (X) e rótulos (y) e armazenados em matrizes do Numpy:

```
# Converte os conjutos de teste e treinamento em arrays (X e y)
X_test = np.array(testdata.iloc[:,0:11])
y_test = np.array(testdata.iloc[:,11])
X_train = np.array(traindata.iloc[:,0:11])
y_train = np.array(traindata.iloc[:,11])
```

Encerrando a preparação, é feita a normalização dos dois conjuntos de dados. O conjunto de teste é normalizado utilizando os mesmos parâmetros (a mesma norma) utilizados no conjunto de treinamento.

```
# Normaliza as features do conjunto de treinamento
X_train_prep = prep.normalize(X_train, axis = 0, return_norm=True)
X_train_normalizd = X_train_prep[0]
X_train_norm = X_train_prep[1]
# Normaliza as features do conjunto de teste
X_test_normalizd = np.divide(X_test, X_train_norm)
```

Preparados os dados, é possível treinar a rede e isso é feito nas linhas seguintes do código, onde a classe MLPClassifier é utilizada para configurar e treinar uma rede neural. Adicionalmente, registramos no vetor loss_curve os valores de custo obtidos pelo algoritmo a cada iteração durante o treinamento para visualização posterior.

No exemplo abaixo, são utilzados os parâmetros que apresentaram a melhor acurácia na predição.

```
# Treinamento do modelo (1)
clf = MLPClassifier(solver='adam',
                    alpha=0.2, # taxa de regularizacao
                    learning_rate_init=0.005, # taxa de aprendizado
                    learning_rate='adaptive', # taxa de aprendizado
                    hidden_layer_sizes=(60,),
                    max_iter=300)
clf.fit(X_train_normalizd, y_train)
loss_curve = clf.loss_curve_
```

Com o modelo ajustado, é possível fazer as predições. Para isso, foi aplicado o método predict() da classe MLPClassifier sobre o conjunto normalizado de dados de teste (X_test_normalizd). O resultado da predição para cada exemplo do conjunto de testes é armazenado no vetor y_pred_test:

```
# Predicoes utilzando conjunto de teste
y_pred_test = clf.predict(X_test_normalizd)
```

A função confusion_matrix(), do módulo metrics do Scikit-Learn, é utilizada para para testar a acurácia das predições do modelo. Ela retorna uma matriz com as quantidades de predições corretas de cada classe, falsos positivos e falsos negativos. No código reproduzido abaixo, esses valores são armazenados na matriz cm.

```
cm = confusion_matrix(y_test, y_pred_test)
```

Para encerrar o script, a função plot_learning_curve() é chamada para exibir dois gráficos com os resultados do processamento. Através deles, foi possível avaliar os diversos resultados obtidos com diferentes combinações de hiperparâmetros.

O primeiro gráfico mostra a taxa de erros obtida a cada iteração do algoritmo durante ajuste da rede neural, armazenada previamente no vetor loss_curve. Esse gráfico não foi solicitado no enunciado do exercício, mas preferi incluí-lo no relatório, porque através dele, foi possível observar que, dependendo das taxas regularização e de aprendizado utilizadas, o algoritmo podia não convergir nunca ou convergir muito depressa, mas sem gerar um modelo eficiente.

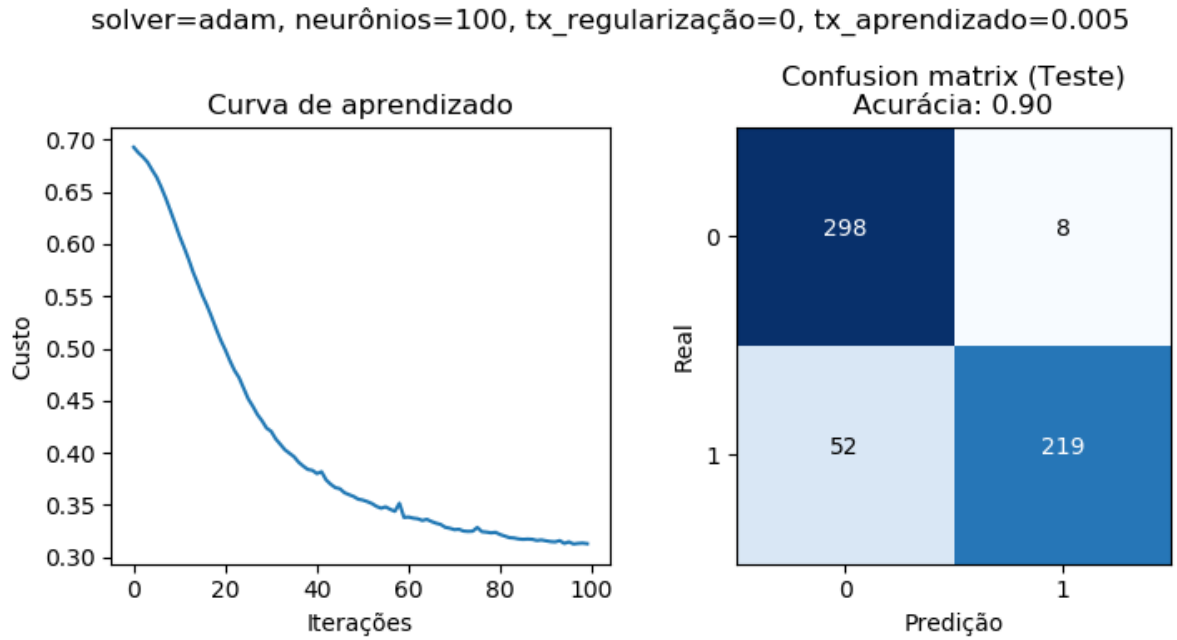
O segundo gráfico, mostra a matriz de confusão, que possibilita avaliar a acurácia do modelo dados os hiperparâmetros informados.

Segue abaixo o código completo da função:

```
def plot_learning_curve(loss_curve, cm, titulo):
    acc = (cm[0,0] + cm[1,1]) / np.sum(cm)
    ## Figura
    plt.figure(figsize=[9,4])
    plt.suptitle(titulo)
    plt.subplot(121)
    plt.title('Curva de aprendizado (Treinamento)')
    plt.xlabel('Iteracoes')
    plt.ylabel('Custo')
    plt.plot(loss_curve, label='loss')

    plt.subplot(122)
    # Configura os rotulos dos eixos
    plt.xticks([0,1],['0','1'])
    plt.yticks([0,1],['0','1'])
    plt.xlabel('Predicao')
    plt.ylabel('Real')
    # Configura os valores de cada classe
    thresh = cm.max()/2  # limite usado para definir quando o texto sera claro ou escuro
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, "{:,}".format(cm[i, j]), horizontalalignment="center",color="white" if cm[i
    plt.title('Confusion matrix (Teste) - Acuracia: %4.2f' % acc)
    # Configura as cores do grafico
    cmap = plt.get_cmap('Blues')
    plt.imshow(cm, cmap=cmap)

    plt.subplots_adjust(top=0.8)
    plt.show()
```

1.3 Resultados

A imagem a seguir mostra os melhores resultados produzidos pelo modelo:

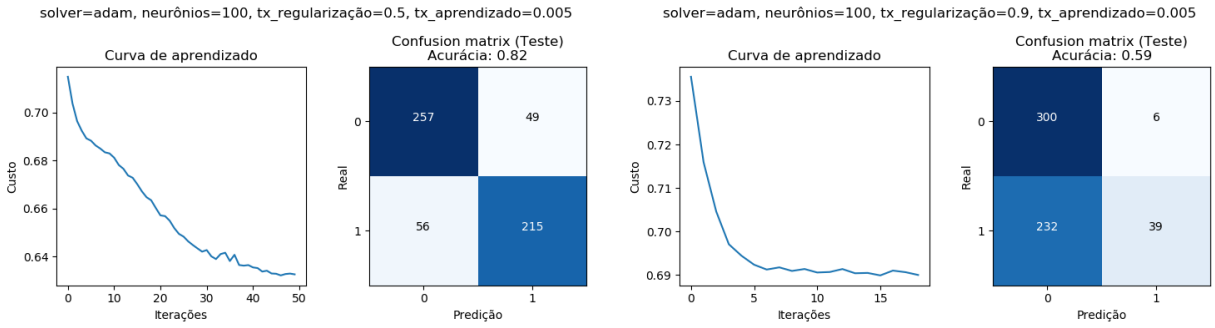


Antes de chegar a esse resultado, diversas combinações de hiperparâmetros foram testadas. Os gráficos abaixo mostram o efeito de utilizar valores muito baixos ou muito altos de parâmetros sobre o ajuste e acurácia do modelo:

1.3.1 Taxa de regularização

Com uma taxa de regularização muito baixa (0 ou próximo de 0) o modelo alcança a melhor acurácia. Acredito que devido a sua simplicidade, a regularização causa uma simplificação excessiva, que acaba prejudicando o desempenho geral.

A taxa de regularização muito alta (próxima ou acima de 1) faz com que todas as predições do modelo caiam em uma única classe (0), como pode ser visto na confusion matrix.



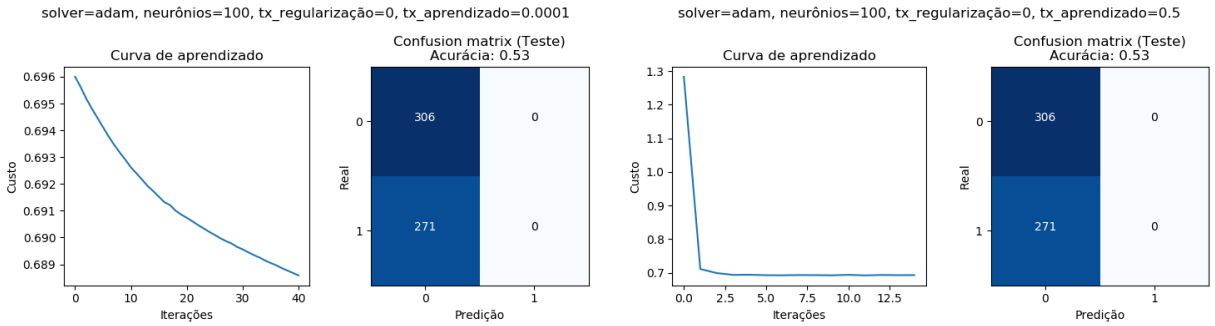
(a) regularização = 0.5 (b) regularização = 0.9

Figura 1.1: Variações na taxa de regularização

1.3.2 Taxa de aprendizado

O MLPClassifier permite a escolha de uma taxa de aprendizado inicial e também métodos para sua atualização durante o processo de minimização. Para os testes, foi utilizada a taxa de aprendizado do tipo constante (learning_rate='constant'), que se comporta como as que já implementamos em outros trabalhos.

Quando é escolhido um valor muito baixo ou muito alto, não é possível obter um bom ajuste do modelo



(a) Muito baixa (b) Muito alta

Figura 1.2: Variações na taxa de aprendizado

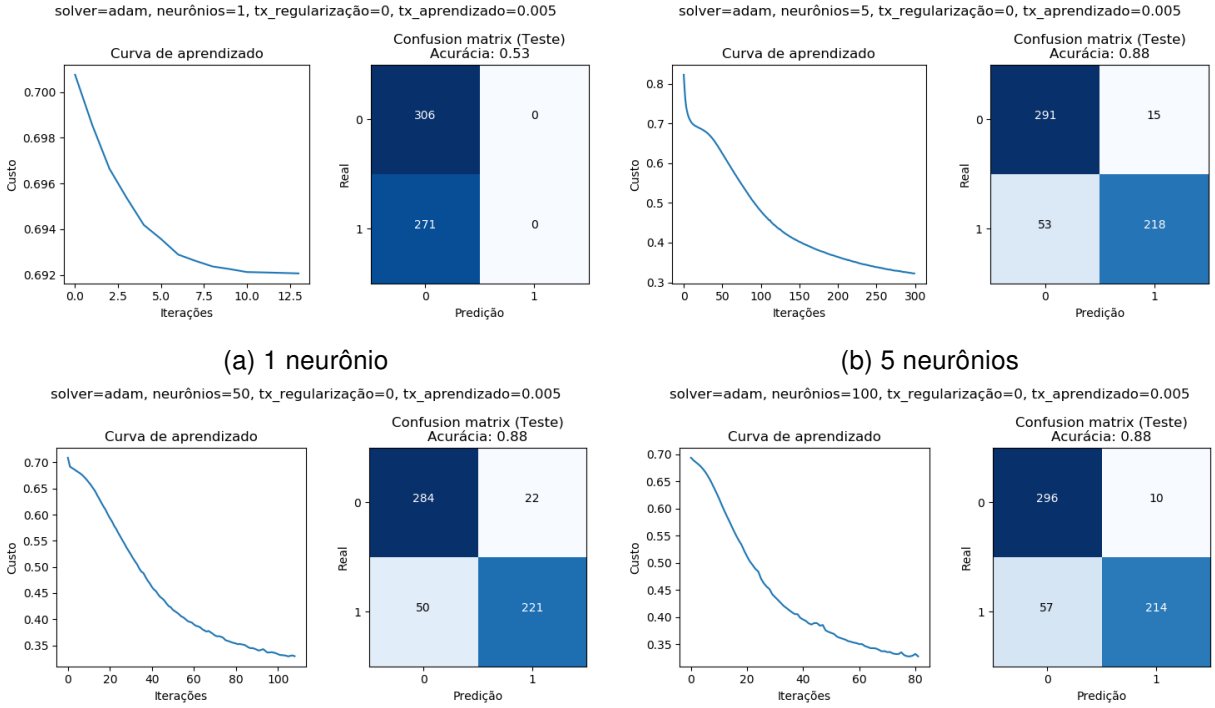
1.3.3 Quantidade de neurônios

Para o problema proposto foi utilizada apenas uma camada oculta e diversas quantidades de neurônios foram testadas.

Com menos de 11 neurônios, foram obtidos resultados irregulares, às vezes não sendo possível convergir dentro do limite de 300 iterações do algoritmo, às vezes convergindo rapidamente, mas sem obter bons resultados no teste.

Em testes com quantidade acima de 11 neurônios, foi possível obter um bom desempenho do classificador, e não foram observadas melhoras significativas no desempenho do classificador à medida que mais neurônios foram utilizados na camada oculta.

Nas imagens a seguir, podem-se observar os resultados obtidos com diversas quantidades de neurônios:



(a) 1 neurônio

(b) 5 neurônios

(c) 50 neurônios

(d) 100 neurônios

Figura 1.3: Variações na quantidade de neurônios

Parte 2 - Redes Convolucionais

O objetivo dessa parte do trabalho é treinar duas redes neurais para classificação de imagens. As redes deverão identificar, dentre as imagens no dataset fornecido, aquelas que contém figuras de gatos.

Desta vez, utilizei o Jupyter Notebook. Todo o código desenvolvido pode ser encontrado no arquivo **Trabalho 4 de AM - Parte 2.ipynb**.

Além do dataset, foi fornecida uma função que carrega as imagens do arquivo para arrays numpy. Essa função e também o código adicional que escrevi para preparar os dados para utilização pelas duas redes são examinados a seguir.

2.1 Carga e preparação dos dados

A função load_dataset() carrega os dados a partir de arquivos de dados HDF5 utilizando a biblioteca h5py. Ao final de sua execução, ela retorna 5 arrays numpy com os conteúdos abaixo:

- train_set_x_orig - exemplos do conjunto de treinamento
- train_set_y_orig - rótulos do conjunto de treinamento
- test_set_x_orig - exemplos do conjunto de teste
- test_set_y_orig - rótulos do conjunto de teste
- classes - descrição das duas classes: "non-cat"e "cat"

Inicializei 5 arrays para armazenar os dados retornados pela função load_dataset():

```
# Carregar as imagens nos conjuntos de treinamento e teste
train_x, train_y, test_x, test_y, classes = load_dataset()
```

Com o trecho de código abaixo, foi possível verificar que as imagens são coloridas e têm o formato 64 x 64 pixels,

```
# Dimensoes das imagens
print(np.shape(train_x))
```

Também foi possível exibir uma amostra:

```
# Visualizar amostra de imagens
plt.imshow(train_x[1])
print(train_y[:,1])
```

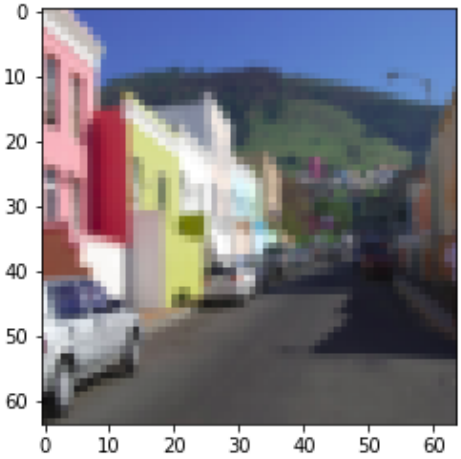


Figura 2.1: Exemplo de imagem do dataset

Em seguida, foram criadas algumas variáveis para armazenar dimensões dos conjuntos de dados que serão importantes mais adiante ao arquitetar as redes neurais:

```
# Obter dimensoes
train_x_shape = np.shape(train_x)
test_x_shape = np.shape(test_x)
ntrain = train_x_shape[0]
ntest = test_x_shape[0]
nrows = train_x_shape[1]
ncols = train_x_shape[2]
nchannels = train_x_shape[3]
```

Posteriormente, durante a seleção de hiperparâmetros para as redes, ficou claro que também no tratamento de imagens é importante normalizar os dados e por isso foi inserido o trecho de código a seguir:

```
# Normalizacao
train_x = train_x / 255
test_x = test_x / 255
```

Obs. A normalização min-max foi utilizada acima de forma simplificada, pois todos os elementos das matrizes estão na mesma escala que vai de 0 a 255.

Isso conclui a parte de tratamento dos dados que é comum às duas redes neurais. A seguir, é analisado o código da primeira delas.

2.2 Rede completamente conectada, com uma camada oculta

A primeira tarefa solicitada foi criar uma rede neural simples, com uma única camada oculta, completamente conectada.

Esse tipo de rede, ao contrário das redes convolucionais que veremos mais adiante, não recebe como entrada uma matriz contendo a imagem, mas sim um vetor contendo uma representação sequencial de todos os seus pixels. Desta forma, a primeira atividade necessária é "achatar"a imagem, o que é feito pelo trecho de código abaixo utilizando o método numpy.reshape():

```
# Reformatar os dados (reshape)
train_x_flat = train_x.flatten().reshape(ntrain, nrows * ncols * nchannels)
test_x_flat = test_x.flatten().reshape(ntest, nrows * ncols * nchannels)
```

Uma das características do modelo solicitado é que a camada de saída deveria ser uma softmax de duas unidades. Como consequência, nosso vetor de rótulos também precisou ser reformatado de forma a utilizar duas colunas para representar as classses "non-cat"(0) e "cat"(1) existentes no vetor original.

Essa conversão de vetor em matriz de classes foi obtida com a aplicação do método to_categorical(), do Keras. A matriz resultante apresenta uma coluna para cada classe existente no vetor original e é inicialmente preenchida com zeros. De acordo com o valor de cada elemento no vetor, a coluna correspondente na matriz resultante recebe o valor um.

```
train_y_cat = keras.utils.to_categorical(train_y[0], 2)
test_y_cat = keras.utils.to_categorical(test_y[0], 2)
```

Após a reformatação, ficamos com as seguintes matrizes:

- train_x_flat: (209, 12288)
- test_x_flat: (50, 12288)
- train_y_cat: (209, 2)
- test_y_cat: (50, 2)

O modelo foi definido com as camadas:

1. (Entrada)
2. Totalmente conectada
3. Softmax

A definição de uma camada de entrada de forma explícita foi substituída pela utilização do parâmetro adicional *input_shape* = na primeira camada oculta, como pode ser visto no código abaixo:

```
# Definir o modelo
model = keras.Sequential()
# Camada Fully Connected
model.add(keras.layers.Dense(128, input_shape = (nrows*ncols*nchannels, ), activation='relu'))
# Softmax com duas saidas:
model.add(keras.layers.Dense(2, activation='softmax'))
```

Após a arquitetura geral da rede, é necessário determinar o método de aprendizado que será utilizado. Isso é feito através do método compile(). Para essa rede, os melhores resultados foram obtidos como o algoritimo de otimização *Adam*, e a função de custo *binary_crossentropy*, como no código abaixo:

```
# Configurar o metodo de aprendizado
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

O passo seguinte é fazer o treinamento do modelo. O método fit() do Keras recebe como parâmetros a quantidade de épocas desejadas para o treinamento e os conjuntos de dados de treinamento e validação (opcional). Durante essa fase, é possível ver na tela, ou armazenar para utilização posterior, os valores obtidos pela função de custo e a acurácia a cada ciclo de treinamento.

```
# Ajustar o modelo
model.fit(train_x_flat, train_y_cat, epochs=30, validation_data=(test_x_flat, test_y_cat))
```

Finalizado o ajuste do modelo, ele pode ser utilizado para fazer predições. Em nosso caso, realizei predições para todos os exemplos do conjunto de teste e armazenei o resultado na variável result, que é utiizada para calcular a acurácia do modelo.

A melhor acurácia obtida com esse modelo, depois de diversos testes de configurações de hiperparâmetros, foi de 74%.

O trecho de código abaixo mostra como foram obtidas as predições e a acurácia:

```
# Avaliar o modelo
result = model.predict(test_x_flat)

# Teste de acuracia / Confusion Matrix - Conjunto de Teste
y_true = test_y[0,:]
y_test = np.round(result[:,1])
cm = confusion_matrix(y_true, y_test)
print(cm)
acc = (cm[0,0] + cm[1,1]) / np.sum(cm)
print("Acuracia: ",acc)
```

2.3 Rede Convolucional

A rede convolucional recebe imagens como entrada. Por esse motivo, essa parte do trabalho não utiliza os dados "achatados"como na parte anterior.

A estrutura da rede ficou assim:

1. Entrada (implícita)
2. Camada Convolucional
3. Max pooling
4. Camada Convolucional
5. Max pooling
6. Camada "flattening"
7. Camada Fully Connected
8. Softmax com duas saídas

As camadas convolucionais e max pooling "interpretam"o conteúdo das imagens antes de passar os dados para as camadas posteriores da rede. A camada flattening converte os dados multidimensionais que recebe para uma única dimensão que é em seguida passada para uma camada completamente conecetada e uma softmax, iguais às utilizadas no modelo anterior.

O código e os parâmetros utilizados são reproduzidos abaixo:

```
# Definir o modelo
model = keras.Sequential()
# Camada Convolucional
model.add(keras.layers.Conv2D(64, (3,3), input_shape=(nrows, ncols, nchannels),
                              use_bias=True, activation='relu'))
# Max pooling
model.add(keras.layers.MaxPooling2D(pool_size = (2, 2)))
# Camada Convolucional
model.add(keras.layers.Conv2D(32, (3,3), input_shape=(nrows, ncols, nchannels),
                              use_bias=True, activation='relu'))
# Max pooling
model.add(keras.layers.MaxPooling2D(pool_size = (2, 2)))
# Camada "flattening"
model.add(keras.layers.Flatten())
# Camada Fully Connected
model.add(keras.layers.Dense(128, activation='relu'))
# Softmax com duas saidas:
model.add(keras.layers.Dense(2, activation='softmax'))
```

Diversos algoritimos de aprendizado foram utilizados e os melhores resultados para essa rede foram obtidos com o *Adam*, com a função de custo *binary_crossentropy*, como no código a seguir:

```
# Configurar o metodo de aprendizado
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Apenas 30 épocas de treinamento são necessárias para essa rede obter seu melhor resultado:

```
model.summary()
model.fit(train_x, train_y_cat, epochs=30)
```

Com essa rede, foi possível obter resultados melhores que anteriormente. A acurácia subiu para 84%. O código utilizado para obter as métricas de desempenho é o mesmo utilizado anteriormente:

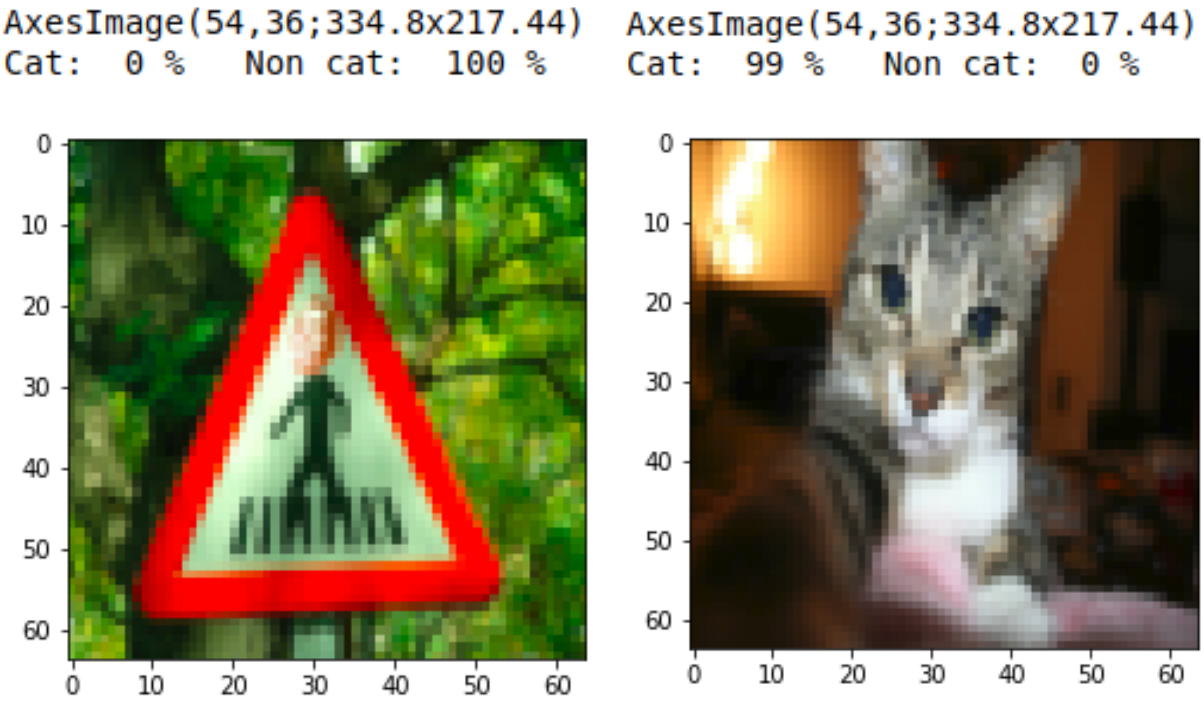
```
# Executa predicoes com os exemplos do conjunto de teste
result = model.predict(test_x)

# Teste de acuracia / Confusion Matrix - Conjunto de Teste
y_true = test_y[0,:]
y_test = np.round(result[:,1])
cm = confusion_matrix(y_true, y_test)
print(cm)
acc = (cm[0,0] + cm[1,1]) / np.sum(cm)
print("Acuracia: ",acc)
```

Ao utilizar nos dois modelos as mesmas duas últimas camadas (totalmente conectada e softmax), com os mesmos parâmentos, foi possível obter a percepção de que a introdução de camadas convolucionais proporciona à rede neural um melhor desempenho na tarefa de classificação de imagens.

2.3.1 Classificação de Imagens

Para finalizar, incluí um trecho de código que permite realizar testes de classificação com as imagens do dataset. Digitando o número de uma das 50 imagens do conjunto de testes (de 0 a 49), obtém-se a classificação ("cat"/"non-car") dada pelo modelo:

```
imagem = 13
test = plt.imshow(test_x[imagem])
print(test)
print("Cat: ",int(result[imagem,1]*100),"%   Non cat: ", int(result[imagem,0]*100),"%" )
```



(a) Non-cat

(b) Cat

Figura 2.2: exemplos de imagens classificadas pelo modelo