Capstone Project

RAJA RANJITH KUMAR ASILETI

Wednesday, February 13, 2019

I. Definition

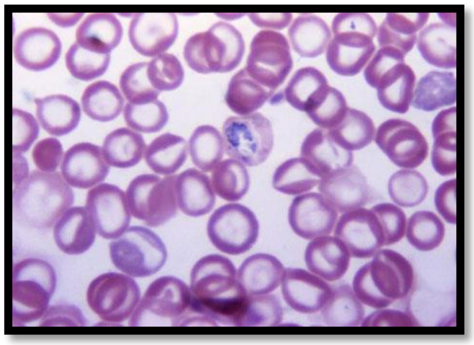
Project Overview

Malaria is a life-threatening disease caused by parasites that are transmitted to people through the bites of infected female Anopheles mosquitoes. It is preventable and curable.

- In 2017, there were an estimated 219 million cases of malaria in 90 countries.
- Malaria deaths reached 435 000 in 2017.
- The WHO African Region carries a disproportionately high share of the global malaria burden. In 2017, the region was home to 92% of malaria cases and 93% of malaria deaths.

Malaria is caused by Plasmodium parasites. The parasites are spread to people through the bites of infected female Anopheles mosquitoes, called "malaria vectors." There are 5 parasite species that cause malaria in humans, and 2 of these species – P. falciparum and P. vivax – pose the greatest threat.

Diagnosis of malaria can be difficult



- Where malaria is not endemic any more (such as in the United States), health-care providers may not be familiar with the disease. Clinicians seeing a malaria patient may forget to consider malaria among the potential diagnoses and not order the needed diagnostic tests.

Laboratories may lack experience with malaria and fail to detect parasites when examining blood smears under the microscope.

- Malaria is an acute febrile illness. In a non-immune individual, symptoms usually appear 10–15 days after the infective mosquito bite. The first symptoms – fever, headache, and chills – may be mild and difficult to recognize as malaria. If not treated within 24 hours, P. falciparum malaria can progress to severe illness, often leading to death.

Microscopic Diagnosis

Malaria parasites can be identified by examining under the microscope a drop of the patient's blood, spread out as a "blood smear" on a microscope slide. Prior to examination, the specimen is stained to give the parasites a distinctive appearance. This technique remains the gold standard for laboratory confirmation of malaria. However, it depends on the quality of the reagents, of the microscope, and on the experience of the laboratories.

Identifying the Malaria detection by using computer Vision architecture is not much accuracy while finding the malaria cells in the human body. So to overcome this problem then Deep learning came into the picture by using it we can identify the uninfected image cell.

Convolutional neural networks have the ability to automatically extract features and learn filters. In previous machine learning solutions, features had to be *manually* programmed in—for example, size, color, the morphology of the cells. Utilizing Convolutional neural networks (CNN) will greatly speed up prediction time while mirroring (or even exceeding) the accuracy of clinicians.

Reference Link:

https://towardsdatascience.com/detecting-malaria-using-deep-learning-fd4fdcee1f5a

Applications

- Save humans by detecting and deploying Image Cells that contain Malaria or not!

- We can extend this project for the several cell detecting diseases to identify with good accuracy.
- Can be used for research purpose on different disease to identify easily.
- We get around 85% of accuracy in this project, we can improve this accuracy by applying much more techniques and also huge cell image to get good performance.

In this project we are considering dataset that contain with two folders called

1. Parasitized (Infected)
2. Uninfected

And a total of **27,558** image of Dimension **148 x 148**

Problem Statement

The Aim of this project is to detect the Malaria by using the Cell Image Dataset. In this project I am going to use PyTorch by employing with the network called Model ResNet 50 (Transfer Learning) for improving the Accuracy for detecting the cell Image.

Metrics

I've been exploring the PyTorch neural network library. When using any library to perform classification, at some point you want to know the classification accuracy. The CNTK and Keras libraries have built-in accuracy functions, but PyTorch (and TensorFlow) do not.

I set out to determine how to compute classification accuracy in PyTorch. Cutting to the chase, the very large number of details that had to be dealt with was really, really surprising to me.

Why I have chosen only accuracy calculation for my problem, because there only two categories which I need get output whether the image is Parasitized and Uninfected. So I thought it is better to use Accuracy rather than F1 score. Strongly I recommended selecting Accuracy as my Metric. As per my dataset there are only two images that I need to classify, so better to use Accuracy is my explanation what I was thinking on it.

Accuracy = 100 * correct / Total

II. Analysis

Data Exploration

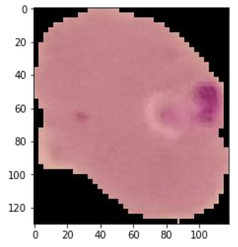
The dataset contains 2 folders:

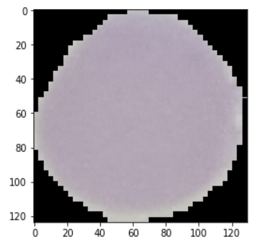
3. Infected
4. Uninfected

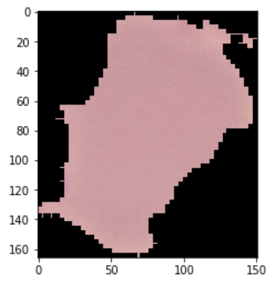
And a total of **27,558** image

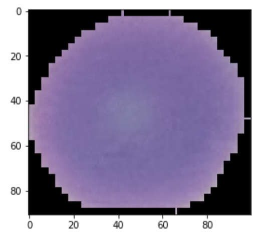
- **Dimension 148 x 148**

Reference Link:
https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria


Parasitized


Uninfected


Uninfected


Uninfected

- PyTorch allow us to train on many variations of the original images that are cropped in different ways or rotated in different ways.
- I want to set up the data transformations for each set of data. In general, we want to have the same types of transformations on the validation and test sets of data. However, with the training data, we can create a more robust model by training it on rotated, flipped, and cropped images.

Exploratory Visualization

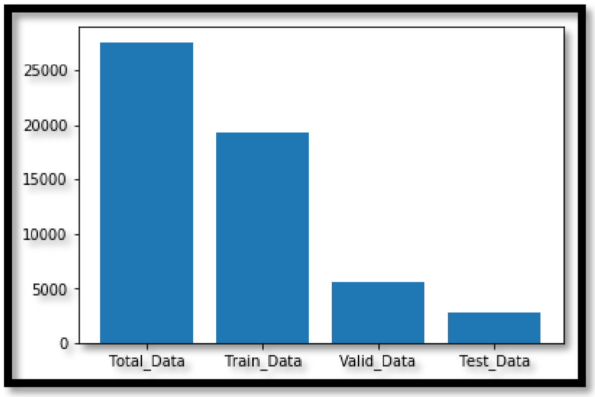
There are 27,558 total images

There are 19,291 Training images

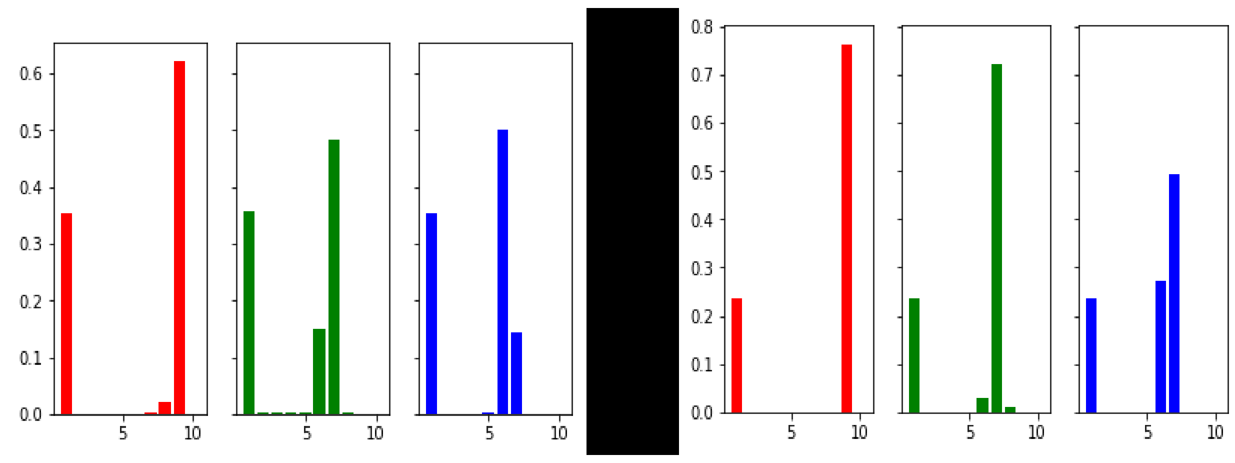
There are 5,511 Validation images

There are 2,756 Test images

<u>The data is divided into Train, Valid and Test</u>



I used RGB intensity histogram for one of image in Parasitized and Uninfected cell images as reviewer's guidance. I new to this technique PyTorch but I tried my best to plot it.



PARASITIZED CELL UNIEFECTED CELL

Reference Link:

https://pytorch.org/docs/stable/torchvision/transforms.html#torchvision.transforms.Compose

Algorithms and Techniques

Deep learning is a subfield of machine learning with algorithms inspired by the working of the human brain. These algorithms are referred to as artificial neural networks. Examples of these neural networks include Convolutional Neural Networks that are used for image classification, Artificial Neural Networks and Recurrent Neural Networks.

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) are everywhere. It is arguably the most popular deep learning architecture. The recent surge of interest in deep learning is due to the immense popularity and effectiveness of convnets. The interest in CNN started with AlexNet in 2012 and it has grown exponentially ever since. In just three years, researchers progressed from 8 layer AlexNet to 152 layer ResNet.

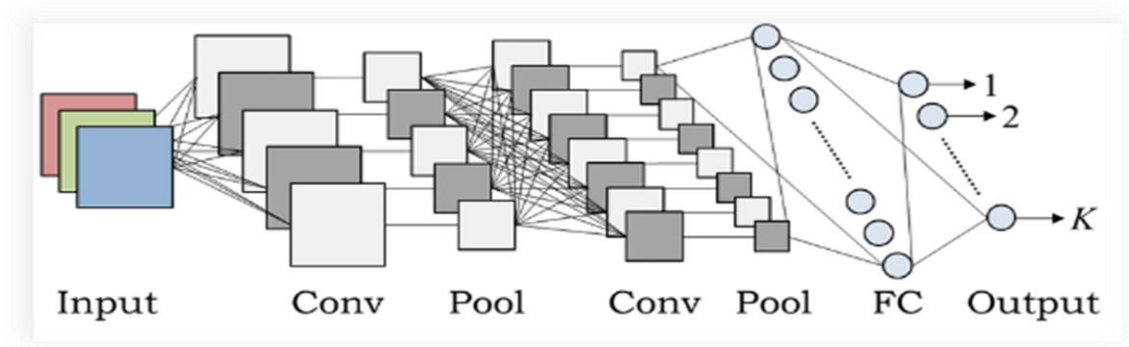
CNN is now the go-to model on every image related problem. In terms of accuracy they blow competition out of the water. It is also successfully applied to recommender systems, natural language processing and more. The main advantage of CNN compared to its predecessors is that it automatically detects the important features without any human supervision. For example, given many pictures of cats and dogs it learns distinctive features for each class by itself.

CNN is also computationally efficient. It uses special convolution and pooling operations and performs parameter sharing. This enables CNN models to run on any device, making them universally attractive.

All in all this sounds like pure magic. We are dealing with a very powerful and efficient model which performs automatic feature extraction to achieve superhuman accuracy (yes CNN models now do image classification better than humans). Hopefully this article will help us uncover the secrets of this remarkable technique.

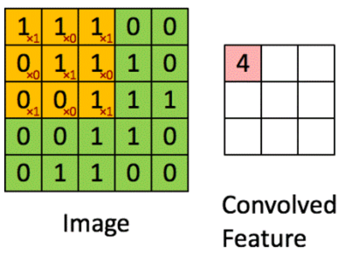
Architecture

There is an input image that we're working with. We perform a series convolution + pooling operations, followed by a number of fully connected layers. If we are performing multiclass classification the output is softmax. We will now dive into each component.



Convolution

The main building block of CNN is the convolutional layer. Convolution is a mathematical operation to merge



two sets of information. In our case the convolution is applied on the input data using a *convolution filter* to produce a *feature map*. There are a lot of terms being used so let's visualize them one by one.

On the left side is the input to the convolution layer, for example the input image. On the right is the convolution *filter*, also called the *kernel*, we will use these terms interchangeably. This is called a *3x3 convolution* due to the shape of the filter.

We perform the convolution operation by sliding this filter over the input. At every location, we do element-wise matrix multiplication and sum the result. This sum goes into the feature map. The green area where the convolution operation takes place is called the *receptive field*. Due to the size of the filter the receptive field is also 3x3.
Here the filter is at the top left, the output of the convolution operation "4" is shown in the resulting feature map. We then slide the filter to the right and perform the same operation, adding that result to the feature map as well.

Pooling

After a convolution operation we usually perform *pooling* to reduce the dimensionality. This enables us to reduce the number of parameters, which both shortens the training time and combats over fitting. Pooling layers down sample each feature map independently, reducing the height and width, keeping the depth intact.

The most common type of pooling is *max pooling* which just takes the max value in the pooling window. Contrary to the convolution operation, pooling has no parameters. It slides a window over its input, and simply takes the max value in the window. Similar to a convolution, we specify the window size and stride.

In CNN architectures, pooling is typically performed with 2x2 windows, stride 2 and no padding. While convolution is done with 3x3 windows, stride 1 and with padding.

Hyperparameters

Let’s now only consider a convolution layer ignoring pooling, and go over the hyperparameter choices we need to make. We have 4 important hyperparameters to decide on:

- **Filter size**: we typically use 3x3 filters, but 5x5 or 7x7 are also used depending on the application. There are also 1x1 filters which we will explore in another article, at first sight it might look strange but they have interesting applications. Remember that these filters are 3D and have a depth dimension as well, but since the depth of a filter at a given layer is equal to the depth of its input, we omit that.
- **Filter count**: this is the most variable parameter; it’s a power of two anywhere between 32 and 1024. Using more filters results in a more powerful model, but we risk overfitting due to increased parameter count. Usually we start with a small number of filters at the initial layers, and progressively increase the count as we go deeper into the network.
- **Stride**: we keep it at the default value 1.
- **Padding**: we usually use padding.

Implementation

Structurally the code looks similar to the ANN we have been working on. There are 4 new methods we haven’t seen before:

- ***Conv2D***: this method creates a convolutional layer. The first parameter is the filter count, and the second one is the filter size. For example in the first convolution layer we create 32 filters of size 3x3. We use *relu*non-linearity as activation. We also enable padding. In Keras there are two options for padding: *same* or *valid*. Same means we pad with the number on the edge and valid means no padding. Stride is 1 for convolution layers by default so we don’t change that. This layer can be customized further with additional parameters; you can check the documentation here.
- ***MaxPooling2D***: creates a maxpooling layer, the only argument is the window size. We use a 2x2 window as it’s the most common. By default

stride length is equal to the window size, which is 2 in our case, so we don't change that.

- ***Flatten*:** After the convolution + pooling layers we flatten their output to feed into the fully connected layers
- ***Dropout***: Dropout is used to prevent overfitting and the idea is very simple. During training time, at each iteration, a neuron is temporarily "dropped" or disabled with probability *p*. This means all the inputs and outputs to this neuron will be disabled at the current iteration.

Introduction to PyTorch

PyTorch is a Python machine learning package based on Torch, which is an open-source machine learning package based on the programming language Lua. PyTorch has two main features:

- Tensor computation (like NumPy) with strong GPU acceleration
- Automatic differentiation for building and training neural networks

Why I preferd PyTorch to other Python deep learning libraries

There are a few reason you might prefer PyTorch to other deep learning libraries:

1. Unlike other libraries like TensorFlow where you have to first define an entire computational graph before you can run your model, PyTorch allows you to define your graph dynamically.
2. PyTorch is also great for deep learning research and provides maximum flexibility and speed.

PyTorch Tensors

PyTorch Tensors are very similar to NumPy arrays with the addition that they can run on the GPU. This is important because it helps accelerate numerical computations, which can increase the speed of neural networks by 50 times or greater. In order to use PyTorch, you'll need to head over tohttps://PyTorch.org/ and install PyTorch.

In order to define a PyTorch tensor, start by importing the torch package. PyTorch allows you to define two types of tensors—a CPU and GPU tensor. For this I am running on GPU machine.

PyTorch Autograd

PyTorch uses a technique called **automatic differentiation** that numerically evaluates the derivative of a function. Automatic differentiation computes backward passes in neural networks. In training neural networks weights are randomly initialized to numbers that are near zero but not zero. **Backward pass** is the process by which these weights are adjusted from right to left, and a forward pass is the inverse (left to right).
torch.autograd is the library that supports automatic differentiation in PyTorch. The central class of this package is torch. Tensor**.** To track all operations on it, set .requires grad as True**.** To compute all gradients, call .backward()**.** The gradient for this tensor will be accumulated in the **.**grad attribute.
If you want to detach a tensor from computation history, call the **.**detach()function. This will also prevent future computations on the tensor from being tracked. Another way to prevent history tracking is by wrapping your code with torch.no_grad():
The Tensor and Function classes are interconnected to build an acyclic graph that encodes a complete history of the computation. The .grad_fnattribute of the tensor references the Function that created the tensor. To compute derivatives, call .backward() on a Tensor. If the Tensor contains one element, you don't have to specify any parameters for the backward()function. If the Tensor contains more than one element, specify a gradient that's a tensor of matching shape.
As an example, you'll create two tensors, one with requires_grad as Trueand the other as False. You'll then use these two tensors to perform addition and sum operations. Thereafter, you'll compute the gradient of one of the tensors.

PyTorch nn Module

This is the module for building neural networks in PyTorch. nn depends on autograd to define models and differentiate them. Let's start by defining the procedure for **training a neural network**:

1. Define the neural network with some learnable parameters, referred to as weights.
2. Iterate over a dataset of inputs.
3. Process input through the network.
4. Compare predicted results to actual values and measure the error.
5. Propagate gradients back into the network's parameters.
6. Update the weights of the network using a simple update rule:

- I used torch.nn.Linear applies a linear transformation to the incoming data

Benchmark

A well-designed convolutional neural network using PyTorch should be able to beat the random choice baseline model easily considering even the Keras model clearly surpasses the initial benchmark. However, due to computational costs, even though applying the transfer learning model with ResNet50 architecture for sufficient number of epochs so that it may be able to converge.

I declared bench mark model with Keras tried to improve the accuracy with the help Transfer learning ResNet50 which is a pertained model. Due to less data Images and also tried to implement to explore for new technique called PyTorch, and I Learned and tried my best to reach the accuracy of 85% which very close to Keras model which done by a kaggle competitor got 94%. So h I was happy that I tried my best to beat this with the new technique called PyTorch which new to me.

III. Methodology

Data Preprocessing

It is useful to set up my **data_transformations**. Essentially, the data transformations in PyTorch allow us to train on many variations of the original images that are cropped in different ways or rotated in different ways.

As I am working through the PyTorch portion of this Nano degree program, I learned a few different ways to do this. However, I found that the documentation was particularly useful for the method I implemented.

First, we want to set up the data transformations for each set of data. In general, we want to have the same types of transformations on the validation and test sets of data. However, with the training data, we can create a more robust model by training it on rotated, flipped, and cropped images.

Though the means and standard deviations are provided to normalize the image values before passing to our network, they could also be found by looking at the mean and standard deviation values of the different dimensions of the image tensors.

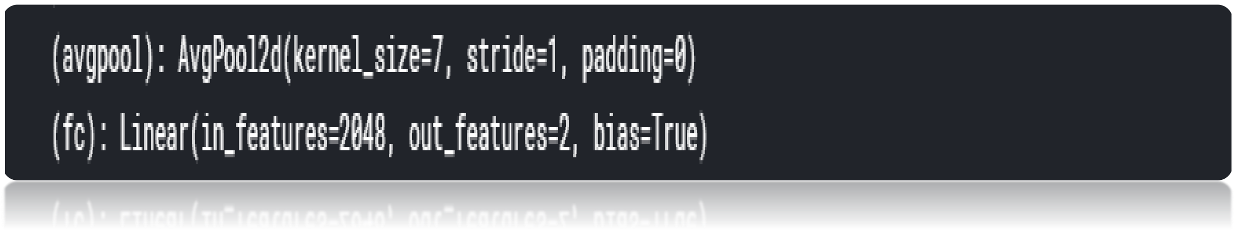
Using **Samplers** and **DataLoader** allows for the ability to easily pass the images through the necessary transformations and then through our network for training or prediction. The code below can easily be reduced or extended to new examples as long as the folder structure provided in the first part of this post is maintained.

Implementation

In recent years, a number of models have been created for reuse in computer vision problems. Using these pre-trained models is known as transfer learning. PyTorch makes it easy to load pre-trained models and build upon them, which is what we will do in this project.

Some of the most popular pre-trained models include **VGGNet, ResNet,** and **AlexNet**, all of which are pre-trained models from the ImageNet Challenge. These pre-trained models allow others to quickly obtain cutting edge results in computer vision without needing the large amounts of compute power, time, and patience in finding the right training technique to optimize the weights.

I decided to use the ResNet50, which we can obtain from the **torchvision** library. However, other models could have been easily used with very similar setup.



```
(avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
(fc): Linear(in_features=2048, out_features=2, bias=True)
```

Model Training

Now that we have our model all set up, we will want to train the final layers. We also want to get an idea of how well it is working! From the same documentation as earlier, we can find a function for training our models. The function shown here is taken nearly verbatim from the documentation.

Let's set up the necessary inputs to this function. There are six inputs to the model:

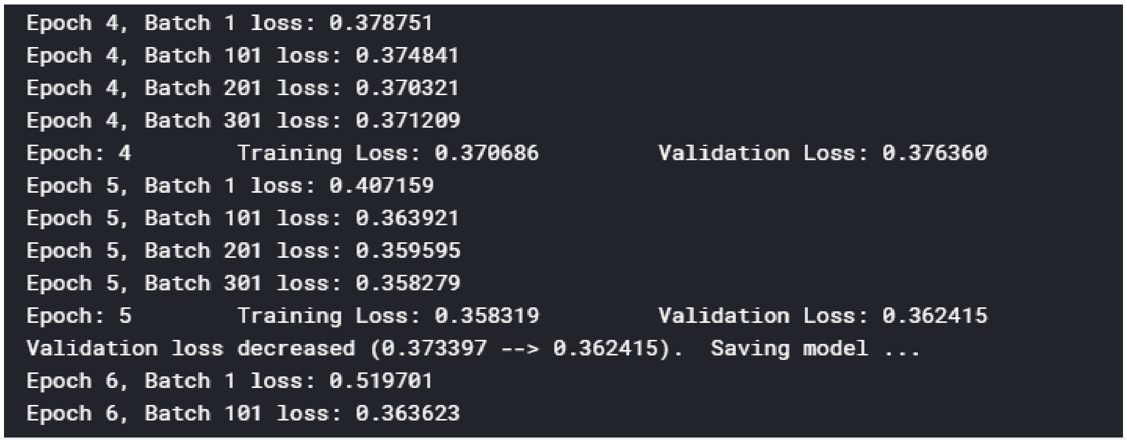
1. The argument is **model**, which is just the model we created in theprevious portion.
2. The argument **criterion** is the method used to evaluate the model fit.
3. The **optimizer** is the optimization technique used to update the weights.

4. The **epoch**, as described in the lessons is full run of feed forward and back propagation through the network.
5. The device was set to **'cuda'** as default, but then could also be set to 'cpu' if you wanted to train your model (for the rest of your life) on your local cpu.
6. The **save_path** is used to save the model if validation loss has decreased.

Kaggel provides time on their classroom GPUs to train my models, and the function I used to train my model was then specified as shown below.

```
train(30, model, optimizer, criterion, use_cuda, 'malaria_detection.pt')
```

If everything is set up correctly, you should see something like the following:



```
Epoch 4, Batch 1 loss: 0.378751
Epoch 4, Batch 101 loss: 0.374841
Epoch 4, Batch 201 loss: 0.370321
Epoch 4, Batch 301 loss: 0.371209
Epoch: 4        Training Loss: 0.370686         Validation Loss: 0.376360
Epoch 5, Batch 1 loss: 0.407159
Epoch 5, Batch 101 loss: 0.363921
Epoch 5, Batch 201 loss: 0.359595
Epoch 5, Batch 301 loss: 0.358279
Epoch: 5        Training Loss: 0.358319         Validation Loss: 0.362415
Validation loss decreased (0.373397 --> 0.362415).  Saving model ...
Epoch 6, Batch 1 loss: 0.519701
Epoch 6, Batch 101 loss: 0.363623
```

This looks promising. The model appears to be learning with each epoch.

Additionally, it doesn't appear that our model is over fitting (at least too much), since the training and validation metrics are not diverging too much.
I found that changing the number of epochs, the optimizer, had the greatest impact on my results.

- **model.state_dict()** holds all of the weights and biases of our model for each layer in a dictionary. **This is the key thing we will need back when want to load our model to use in the future!**

- All of this information is saved to a file. The extension on this file doesn't seem to be very important to the community. I have seen **.pth** suggested by the creator of PyTorch, so I used it below. However, I have also seen **.dat**

```
model.load_state_dict(torch.load('malaria_detection.pt'))
```

Refinement

PyTorch makes it easy to load pre-trained models and build upon them, which is what we will do in this project.

Some of the most popular pre-trained models include **VGGNet, ResNet,** and **AlexNet**, all of which are pre-trained models from the ImageNet Challenge. These pre-trained models allow others to quickly obtain cutting edge results in computer vision without needing the large amounts of compute power, time, and patience in finding the right training technique to optimize the weights.

I decided to use the ResNet50, which we can obtain from the **torchvision** library. However, other models could have been easily used with very similar setup.

IV. Results

Model Evaluation and Validation

I've been explored the PyTorch neural network library. When using any library to perform classification, at some point you want to know the classification accuracy. The CNTK and Keras libraries have built-in accuracy functions, but PyTorch (and TensorFlow) do not.

I set out to determine how to compute classification accuracy in PyTorch. Cutting to the chase,the very large number of details that had to be dealt with was really, really surprising to me .

As we can see that 85% accuracy has been made by using the PyTorch which tried to improve the technique which is new to me and also applied the Transfer learning for this model to get good performance. From this we can conclude that RESNET50 is robust to unseen data. And can be able to generalize the data pretty well.

Instead of using F1 score it is better to use the Accuracy as metric because it all about infected or not based on two categories. So I had chosen Accuracy as a metric which shows result simple and clear.

Finally I am statisfied with my work what I made to get the accuracy above 80%.

Justification

When compared with my benchmark model, my model gives training accuracy up to 84% whereas my testing accuracy is around 85%. The results obtained from my model above satisfactory even though I am not beat the Keras accuracy. But I learned more knowledge to explore new Technology called **PyTorch**.

In deep learning we cannot justify that no model can be good at some certain point of view, It just all about improving the performance by add-on new techniques and changing the hyper parameters which will be generate a good outcome that gives some satisfactory to my work.

Now I can confidently say that my still we can improve the solution significant to solve the problem by adding much more cell image data.
Which model to choose:-
I think there is very close to Keras and PyTorch it head to say which model is best.

But considering the PyTorch it increases the speed of training model when compared to the Keras. Even though I am new to this PyTorch technique I tried to improve the accuracy at some extent.

So exploring to new techniques is good, but getting good accuracy is also matters. 85% is not a bad accuracy it should be consider to be a good model that calssifies the image whether uninfected or Not.

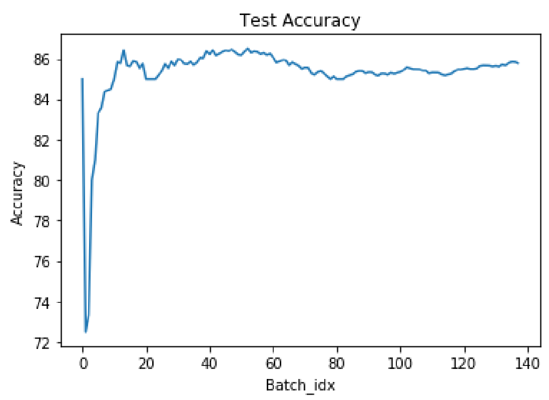
V. Conclusion

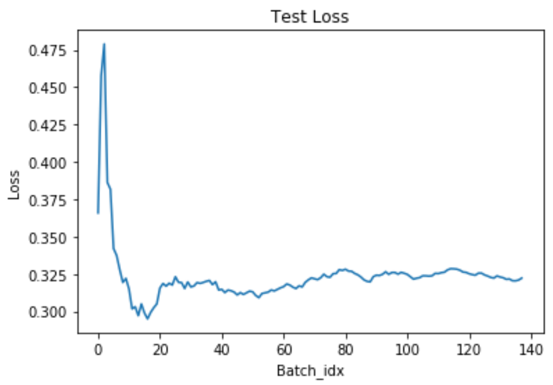
Free-Form Visualization

Visualizing the accuracy is can be done by using a package called visdom for PyTorch but it is not available in kaggle and my system also not supporting that packages to install. So due to time constraint I am trying to explain that this project has been successfully achieved a good accuracy of 85%, but it is very close to benchmark model which I chosen.

I think there is very close to Keras and PyTorch it head to say which model is best.

But considering the PyTorch it increases the speed of training model when compared to the Keras. Even though I am new to this PyTorch technique I tried to improve the accuracy at some extent.





Reflection

In this capstone project I have taken image classification as my thought of interest inspired me to inspiring the world day by day which is called Deep Learning.

In this process I have learned many things.
1) First thing I have learned about data retrieval processes. When I am doing research about retrieval

2) I have also learned how to use kaggle kernals and how to commit it and reproduce my work.
3) Then I used my skills on representation of overall number of images, and also category wise in both
training and testing images by using Matplotlib library. I have realized that its most informative in beginning in understanding size of your project.
4) When I Started Transforming the data all the parameter are comes under in single line for resize, normalization without write all theses explicitly again and again.
5) I am also learn how to use Transfer Learning Technique to improve the Training model.
6) Then I learned few techniques which are been used in PyTorch, but not more still need to improve more knowledge on PyTorch to get good accuracy on this project.

7) I learn that Image resizing is more important in image classification because we can’t expect every image of same size.

8) Then I learned that How we divide the data into training and validation using traint_test_split method.
9) Last but not least without visualizing results we can't trust the robustness of a model. Problems?

Improvement

PyTorch's popularity is driven in large part by a more gentle learning curve compared to TensorFlow. In particular, PyTorch adopted a more dynamic and

TensorFlow has a much larger ecosystem, with important supporting tools like the excellent Keras, a simplified API that makes TensorFlow much more consumable, and TensorBoard, an impressive visualization tool that makes it easy to plot a wide variety of model metrics.The PyTorch ecosystem isn't standing still though.

Improving my model better than this by using new techniques such as FastAI Which gets more accuracy by training the model, that's sounds good to hear when I heard about FastAI. I will be going to learn the FastAI methods how will be going to use it and will improve considering this project accuracy as benchmark and I will implement the FastAI technique for this problem statement.

FINAL CONCLUSION:

Firstly I want to thank all the members who shared their knowledge by udacity Nano Degree team, I learned huge amount of knowledge that I never learned till now.

Still need to improve my project in future nd want post this project in kaggle. This capstone

project was very interesting and learned a lot and visited many blogs and article to learn this new technique called PyTorch.

Thanks to Udacity Team.