ResNet model on cifar100 dataset technical documentation

UAS Machine Learning

ResNet

- The most popular Deep Convolutional Neural Network architecture design
- Deep Learning learn in hierarchical set of representation such as low, mid, and high level features
- In images it represent shape, edges, and object.
- Theoretically more layers should enrich the level of layer

The Million dollar question is ?

Should adding more layer will enhance the learning process of neural network ?

answer

- In the ResNet paper, it is stated that adding convolutional layer on top of activation and batch normalization gets the model worse not better.
- Higher level layer have a higher error

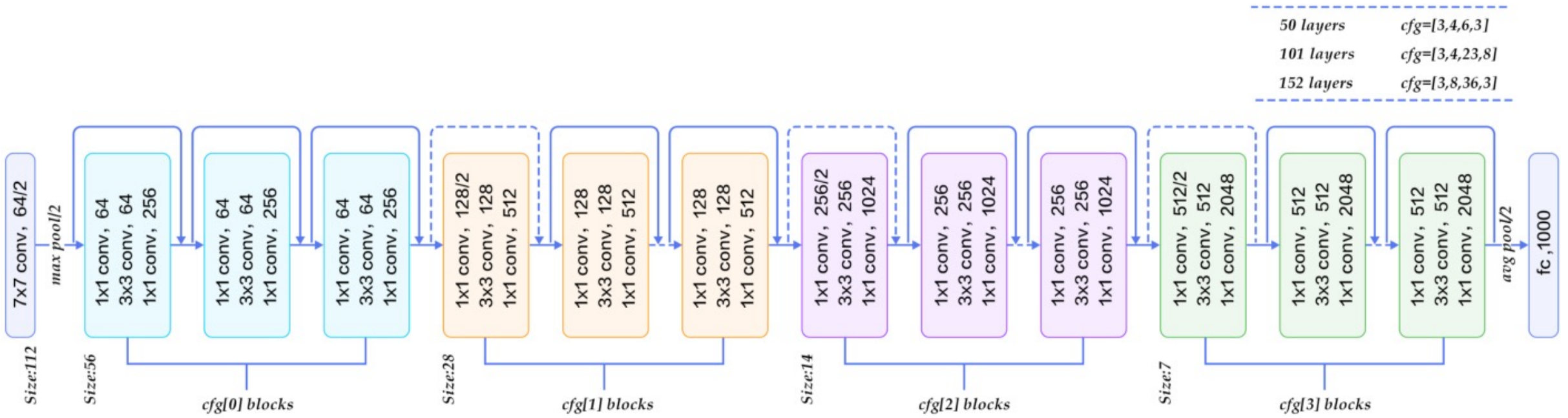
Construction insight

- If you consider a shallow architecture with the deeper counterpart with more layer
- Theoretically the deeper model would just need to copy the shallower model with identity mapping
- The constructed solution suggested that the deeper layer should not have error higher than its shallow counterparts

How ResNet help

- The skip connection in ResNet solve the problem of vanishing gradient in deep neural networks by allowing this alternate shortcut path for the gradient to flow through.
- these connections help by allowing the model to learn the identity functions which ensures that the higher layer will perform at least as good as the lower layer, and not worse

ResNet Architecture



Technical Documentation

```python
class BaseModel(nn.Module):
    def training_step(self,batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out,labels)
        return loss

    def validation_step(self,batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out,labels)
        acc = accuracy(out,labels)
        return {"val_loss":loss.detach(),"val_acc":acc}

    def validation_epoch_end(self,outputs):
        batch_losses = [loss["val_loss"] for loss in outputs]
        loss = torch.stack(batch_losses).mean()
        batch_accuracy = [accuracy["val_acc"] for accuracy in outputs]
        acc = torch.stack(batch_accuracy).mean()
        return {"val_loss":loss.item(),"val_acc":acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {:.4f}, val_loss: {:.4f}, val_acc: {:.4f}".format(
            epoch, result['lrs'][-1], result['train_loss'], result['val_loss'], result['val_acc']))
```

The Base Model

- The Base Model basically will help us keep track of the training process

This the code that inherited from the ResNet architecture

```
def conv_shortcut(in_channel, out_channel, stride):
    layers = [nn.Conv2d(in_channel, out_channel, kernel_size=(1,1), stride=(stride, stride)),
              nn.BatchNorm2d(out_channel)]
    return nn.Sequential(*layers)

def block(in_channel, out_channel, k_size,stride, conv=False):
    layers = None

    first_layers = [nn.Conv2d(in_channel,out_channel[0], kernel_size=(1,1),stride=(1,1)),
                    nn.BatchNorm2d(out_channel[0]),
                    nn.ReLU(inplace=True)]
    if conv:
        first_layers[0].stride=(stride,stride)

    second_layers = [nn.Conv2d(out_channel[0], out_channel[1], kernel_size=(k_size, k_size), stride=(1,1), padding=1),
                    nn.BatchNorm2d(out_channel[1])]

    layers = first_layers + second_layers

    return nn.Sequential(*layers)
```

```
class ResNet(BaseModel):

    def __init__(self, in_channels, num_classes):
        super().__init__()

        self.stg1 = nn.Sequential(
                                nn.Conv2d(in_channels=in_channels, out_channels=64, kernel_size=(3),
                                          stride=(1), padding=1),
                                nn.BatchNorm2d(64),
                                nn.ReLU(inplace=True),
                                nn.MaxPool2d(kernel_size=3, stride=2))

        ##stage 2
        self.convShortcut2 = conv_shortcut(64,256,1)

        self.conv2 = block(64,[64,256],3,1,conv=True)
        self.ident2 = block(256,[64,256],3,1)


        ##stage 3
        self.convShortcut3 = conv_shortcut(256,512,2)

        self.conv3 = block(256,[128,512],3,2,conv=True)
        self.ident3 = block(512,[128,512],3,2)


        ##stage 4
        self.convShortcut4 = conv_shortcut(512,1024,2)

        self.conv4 = block(512,[256,1024],3,2,conv=True)
        self.ident4 = block(1024,[256,1024],3,2)
```

```
    ##Classify
    self.classifier = nn.Sequential(
                                    nn.AvgPool2d(kernel_size=(4)),
                                    nn.Flatten(),
                                    nn.Linear(1024, num_classes))

def forward(self,inputs):
    out = self.stg1(inputs)

    #stage 2
    out = F.relu(self.conv2(out) + self.convShortcut2(out))
    out = F.relu(self.ident2(out) + out)
    out = F.relu(self.ident2(out) + out)
    out = F.relu(self.ident2(out) + out)

    #stage3
    out = F.relu(self.conv3(out) + (self.convShortcut3(out)))
    out = F.relu(self.ident3(out) + out)
    out = F.relu(self.ident3(out) + out)
    out = F.relu(self.ident3(out) + out)
    out = F.relu(self.ident3(out) + out)

    #stage4
    out = F.relu(self.conv4(out) + (self.convShortcut4(out)))
    out = F.relu(self.ident4(out) + out)
    out = F.relu(self.ident4(out) + out)
    out = F.relu(self.ident4(out) + out)
    out = F.relu(self.ident4(out) + out)
    out = F.relu(self.ident4(out) + out)
    out = F.relu(self.ident4(out) + out)

  #Classify
  out = self.classifier(out)#100x1024

  return out
```

Training process

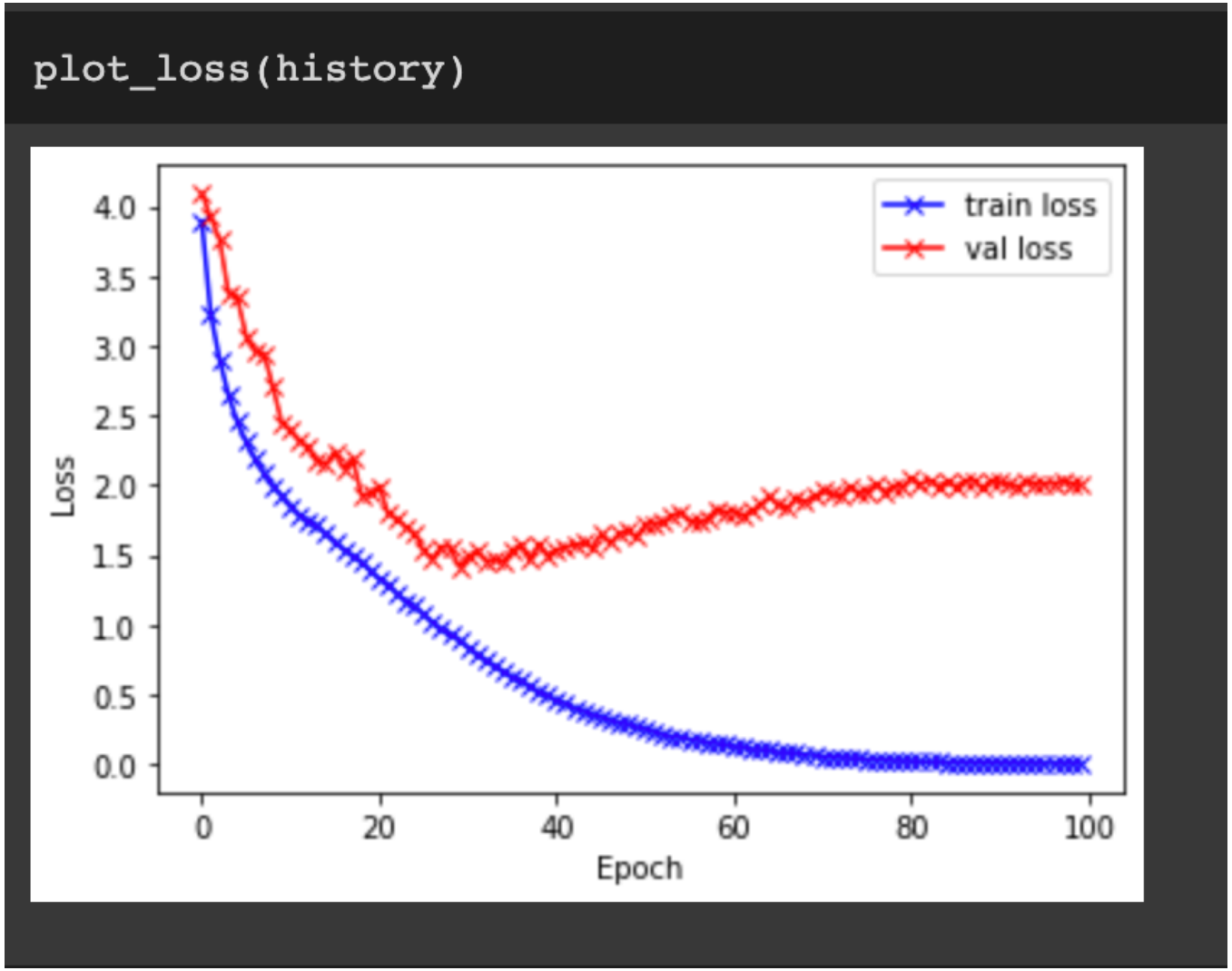
```
epochs = 100
optimizer = torch.optim.Adam
max_lr = 1e-3
grad_clip = 0.1
weight_decay = 1e-5
scheduler = torch.optim.lr_scheduler.OneCycleLR
```

```
%%time
history = fit(epochs=epochs, train_dl=train_dl, test_dl=test_dl, model=model,
              optimizer=optimizer, max_lr=max_lr, grad_clip=grad_clip,
              weight_decay=weight_decay, scheduler=torch.optim.lr_scheduler.OneCycleLR)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning: This DataLoader will create 4 worker processes in total. Our suggeste
  cpuset_checked))
Epoch [0], last_lr: 0.00004, train_loss: 3.8882, val_loss: 4.0885, val_acc: 0.0690
Epoch [1], last_lr: 0.00005, train_loss: 3.2250, val_loss: 3.9202, val_acc: 0.1080
Epoch [2], last_lr: 0.00006, train_loss: 2.8995, val_loss: 3.7595, val_acc: 0.1396
Epoch [3], last_lr: 0.00008, train_loss: 2.6441, val_loss: 3.3755, val_acc: 0.1913
Epoch [4], last_lr: 0.00010, train_loss: 2.4513, val_loss: 3.3395, val_acc: 0.2099
Epoch [5], last_lr: 0.00013, train_loss: 2.3065, val_loss: 3.0687, val_acc: 0.2306
Epoch [6], last_lr: 0.00016, train_loss: 2.1825, val_loss: 2.9561, val_acc: 0.2580
Epoch [7], last_lr: 0.00020, train_loss: 2.0817, val_loss: 2.9417, val_acc: 0.2703
Epoch [8], last_lr: 0.00024, train_loss: 1.9940, val_loss: 2.7076, val_acc: 0.3083
Epoch [9], last_lr: 0.00028, train_loss: 1.9250, val_loss: 2.4491, val_acc: 0.3585
Epoch [10], last_lr: 0.00032, train_loss: 1.8515, val_loss: 2.3955, val_acc: 0.3723
Epoch [11], last_lr: 0.00037, train_loss: 1.7879, val_loss: 2.3129, val_acc: 0.3904
Epoch [12], last_lr: 0.00042, train_loss: 1.7444, val_loss: 2.2757, val_acc: 0.3949
Epoch [13], last_lr: 0.00047, train_loss: 1.7072, val_loss: 2.1697, val_acc: 0.4228
Epoch [14], last_lr: 0.00052, train_loss: 1.6455, val_loss: 2.1539, val_acc: 0.4296
Epoch [15], last_lr: 0.00057, train_loss: 1.5871, val_loss: 2.2439, val_acc: 0.4118
Epoch [16], last_lr: 0.00062, train_loss: 1.5353, val_loss: 2.1178, val_acc: 0.4455
Epoch [17], last_lr: 0.00067, train_loss: 1.4943, val_loss: 2.1941, val_acc: 0.4188
```

Validation and train loss

```
plot_loss(history)
```



Predicting the image

```
img, label = test_data[100]
plt.imshow(img.permute(1, 2, 0))
print('Label:', test_data.classes[label], ', Predicted:', predict_image(img, model))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Label: crab , Predicted: crab
```