Mathematical Foundations of Deep Neural Networks, M1407.001200
E. Ryu
Fall 2021

Homework 5
Due 5pm, Friday, October 15, 2021

Problem 1: *Implementing backprop for MLP.* Consider the multi-layer perceptron

$$\begin{aligned} y_L &= A_L y_{L-1} + b_L \\ y_{L-1} &= \sigma(A_{L-1} y_{L-2} + b_{L-1}) \\ &\vdots \\ y_2 &= \sigma(A_2 y_1 + b_2) \\ y_1 &= \sigma(A_1 x + b_1), \end{aligned}$$

where $x \in \mathbb{R}^{n_0}$, $A_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, $b_\ell \in \mathbb{R}^{n_\ell}$, and $n_L = 1$. Let $\sigma(z) = (1+e^{-z})^{-1}$ be the sigmoid activation function. Let $f_\theta(x) = y_L$ and consider the loss function

$$\ell(\theta) = \frac{1}{2}(f_\theta(X_{\text{data}}) - Y_{\text{data}})^2.$$

Download the starter code `mlp_backprop.py` and implement backprop using the gradient computation of homework 4 problem 6. Your code should roughly be of the form:

```
# forward pass
y_list = [X_data]
y = X_data
for ell in range(L):
  S = sigma if ell<L-1 else lambda x: x
  y = S(A_list[ell]@y+b_list[ell])
  y_list.append(y_next)

# backward pass
dA_list = []
db_list = []
dy = y-Y_data
for ell in reversed(range(L)):
  S = sigma_prime if ell<L-1 else lambda x: torch.ones(x.shape)
  A, b, y= A_list[ell], b_list[ell], y_list[ell]
  db = ...    # dloss/db
  dA = ...    # dloss/dA
  dy = ...    # dloss/dy
  dA_list.insert(0,dA)
  db_list.insert(0,db)
```

The starter code provides code that performs gradient computation using autograd. Compare your results against the autograd results.

Problem 2: *Vanishing gradients.* When training very deep neural networks, one often encounters the problem of vanishing gradients. Consider the MLP of homework 4 problem 6. Assume the activation function σ is the sigmoid activation function. Define $\tilde{y}_i = A_i y_{i-1} + b_i$ for $i = 1, \ldots, L$. So $y_L = \tilde{y}_L$ and $y_i = \sigma(\tilde{y}_i)$ for $i = 1, \ldots, L-1$.

Assume the matrices $A_1, \ldots, A_L$ are all not too large. If A_j is small for some $j \in \{\ell+1, \ldots, L\}$, then

$$\frac{\partial y_L}{\partial b_i}, \quad \frac{\partial y_L}{\partial A_i}$$

for $i = 1, \ldots, \ell$ become small. If $\tilde{y}_j$ has large absolute value for some $j \in \{\ell+1, \ldots, L-1\}$, then

$$\frac{\partial y_L}{\partial b_i}, \quad \frac{\partial y_L}{\partial A_i}$$

for $i = 1, \ldots, \ell$ become small. Explain why this is the case.

Clarification. For the purpose of this problem, let's say that a vector or a matrix is "small" if all of its entries are small. Define "large" analogously. Also, (not too large)×(small)=(small).

Remark. Neural networks built with ReLU tend to suffer less from vanishing gradients compared to networks built with sigmoid or tanh. This is because $\sigma'(z) \to 0$ as $z \to \pm\infty$ for sigmoid and tanh, while $\sigma'(z) \to 0$ only as $z \to -\infty$ for ReLU.

Problem 3: *Two forms of momentum SGD.* There are two forms for the (non-Nesterov) momentum SGD. Form I is

$$\theta^{k+1} = \theta^k - \alpha g^k + \beta(\theta^k - \theta^{k-1})$$

for $k = 0, 1, \ldots$, where $\theta^{-1} = \theta^0$. This form is more commonly invoked in mathematical discussions as it makes the "momentum term" $\beta(\theta^k - \theta^{k-1})$ clearly visible. Form II is

$$\begin{aligned} v^{k+1} &= g^k + \beta v^k \\ \theta^{k+1} &= \theta^k - \alpha v^{k+1} \end{aligned}$$

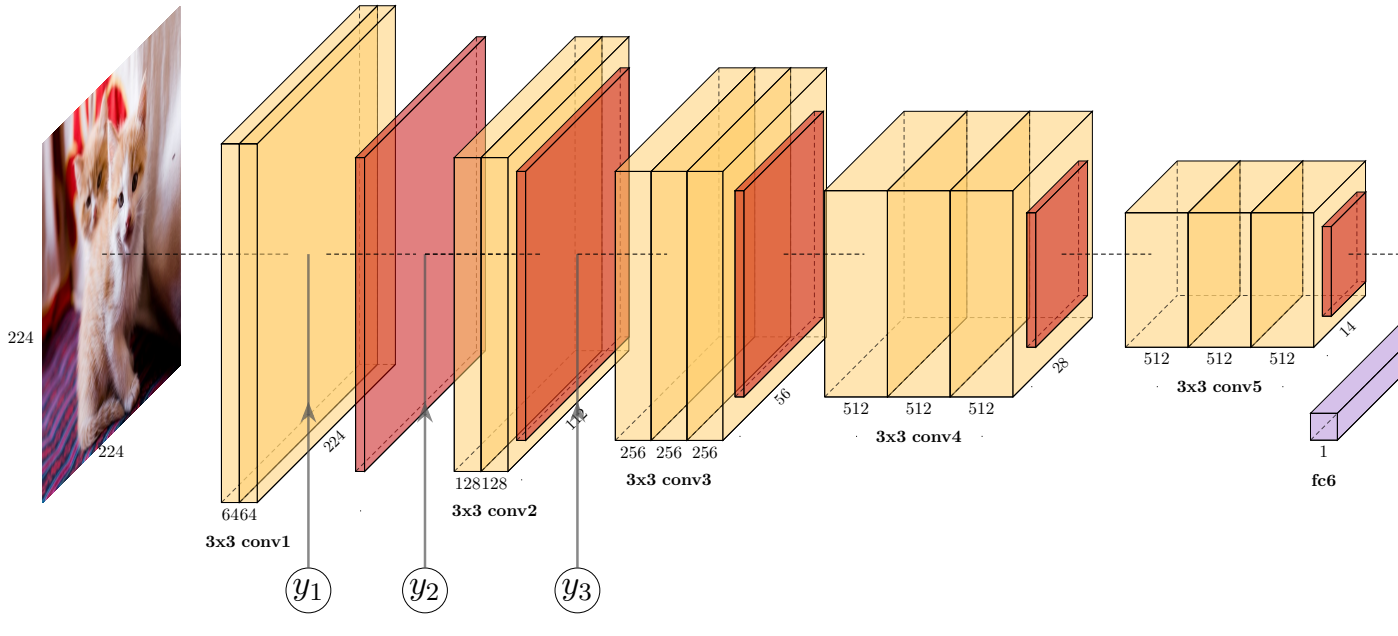
for $k = 0, 1, \ldots$, where $v^0 = 0$. This is the form implemented in PyTorch with the option `Nesterov=false`. Show that the two forms are equivalent in the sense that given a starting point $\theta^0 \in \mathbb{R}^n$ and a sequence of stochastic gradients $g^0, g^1, \ldots \in \mathbb{R}^n$, Forms I and II produce the same $\theta^1, \theta^2, \ldots$ sequence.

Hint. Use induction.

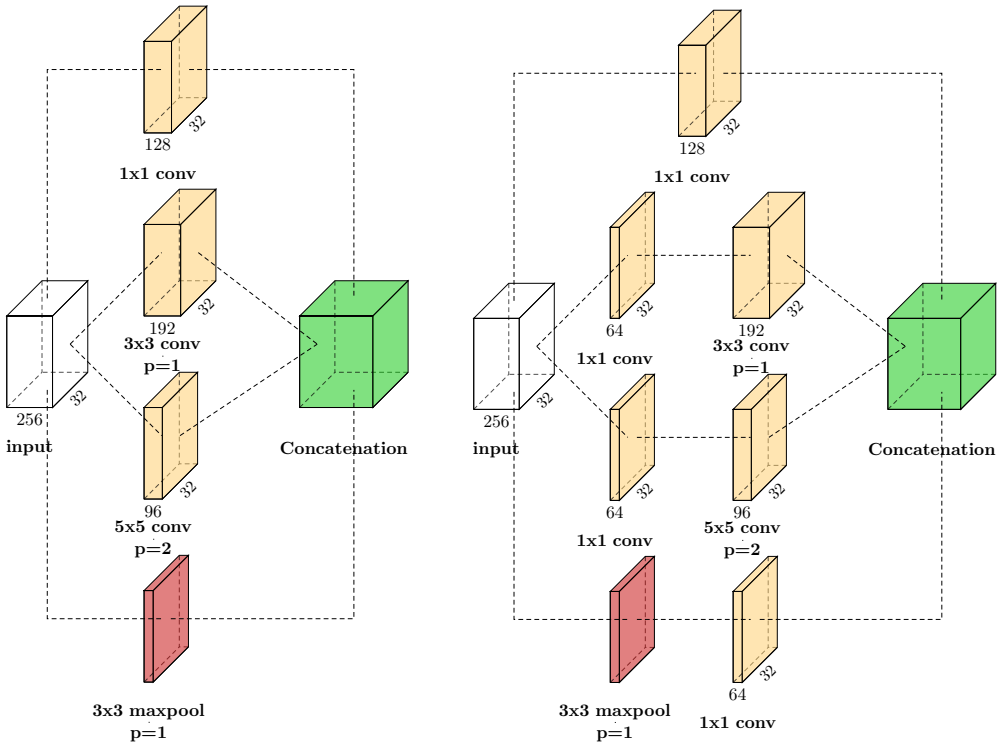
Problem 4: The *receptive field* of a neuron is the set of input pixels the neuron value depends on. Consider the VGG16 network, and consider the intermediate values y_1, y_2, and y_3, which are respectively the outputs of the second convolutional layer, the first maxpool layer, and the second maxpool layer. Assume the input X has a batch size 1 and otherwise has dimensions $3 \times 224 \times 224$. For $p = 1, 2, 3$, describe the receptive field of $y_p[k, i, j]$, i.e., which values of $X[c, m, n]$ does $y_p[k, i, j]$ depend on? Here, m, n and i, j denote the spacial dimensions and c and k denote the channels.

```
class VGG16(nn.Module) :
  def __init__(self) :
    super(VGG16, self).__init__()
    self.conv_layer1 = nn.Sequential(
        nn.Conv2d(3,64,kernel_size=3,padding=1),       #64x224x224
        nn.ReLU(),
        nn.Conv2d(64,64 kernel_size=3,padding=1),       #64x224x224
        nn.ReLU() )
    self.pool1 = nn.MaxPool2d(kernel_size=2,stride=2) #64x112x112
    self.conv_layer2 = nn.Sequential(
        nn.Conv2d(64,128,kernel_size=3,padding=1),    #128x112x112
        nn.ReLU(),
        nn.Conv2d(128,128,kernel_size=3,padding=1),   #128x112x112
        nn.ReLU()   )
    self.pool2 = nn.MaxPool2d(kernel_size=2,stride=2)  #128x56x56
    ...

  def forward(self, x) :
    y1 = self.conv_layer1(x)
    y2 = self.pool2(y1)
    y3 = self.pool2(self.conv_layer2(y2))
    ...
```



Problem 5: Consider the naïve inception module and the inception module with 1×1 "bottleneck" convolutions. Assume the input has 32×32 spatial dimensions and 256 channels. The numbers of output channels of each convolution operation are specified in the figure. Assume all convolutions use biases. A nonlinear activation function is applied after every convolution.



Between the two modules, compare:

(i) the number of trainable parameters and

(ii) the number of additions, multiplications, and activation function evaluations (separately for the three types of operations) required to forward-evaluate the module.

Remark. A more complete investigation in the spirit of part (ii) would count the arithmetic operations of a gradient computation via a backward pass. For the sake of simplicity, we only consider the forward pass.

Clarification. For the purpose of this problem, include the additional operations incurred due to zero-padding. You do not need to compare operations incurred by the maxpool; not only does maxpool not utilize any additions, multiplications, or activation evaluations, but the amount of operations of the maxpools of two modules are identical, so there is no need to compare.

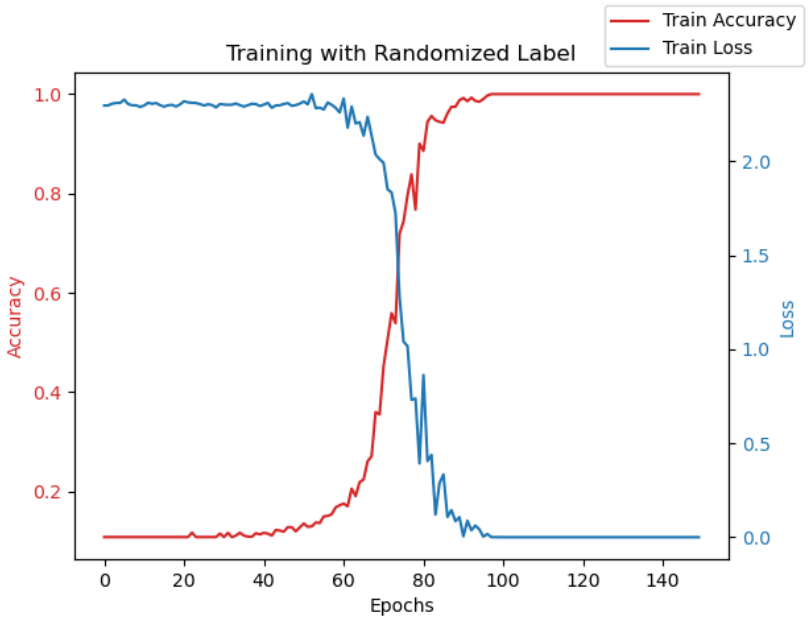
Problem 6: *Large neural networks memorize and interpolate training data.* In 2017, Zhang et al. [1] showed that modern deep neural networks can exactly memorize and interpolate training labels, even when the labels are completely randomized. The experiment considered a 10-class classification problem with (unmodified) data $X_1, \dots, X_N$ and completely randomized labels $Y_i \sim \text{Uniform}\{1, \dots, 10\}$ for $i = 1, \dots, N$. Carry out this experiment with the MNIST data on a variation of the AlexNet architecture as provided in the starter code `label_memorization.py`. Use SGD with learning rate 0.1, batchsize 64, and 150 epochs. To reduce the computation time of this experiment, use only 10% of the MNIST training data, i.e., select a subset of $6,000$ images among the $60,000$ training images.

Clarification. Once Y_i is set to a random class, it should be fixed throughout the training epochs, i.e., do not randomize Y_i again every epoch.

Remark. Of course, the "trained" neural net achieves the generalization performance of 10%.

Remark. This paper by Zhang et al. [1] was highly influential as it shattered all classical statistical expectations. It meant neural networks generalize well despite having the capacity to completely overfit. The insight of this work lead to the formulation of the double descent phenomenon.

Hint. Expect the training to behave as follows:



References

[1] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, Understanding deep learning requires rethinking generalization, *ICLR*, 2017.