Machine Learning Notes

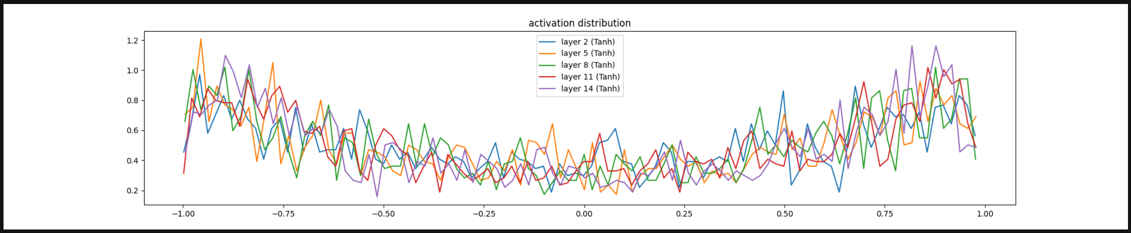
Zach Virgilio

October 20, 2025

Makemore notes

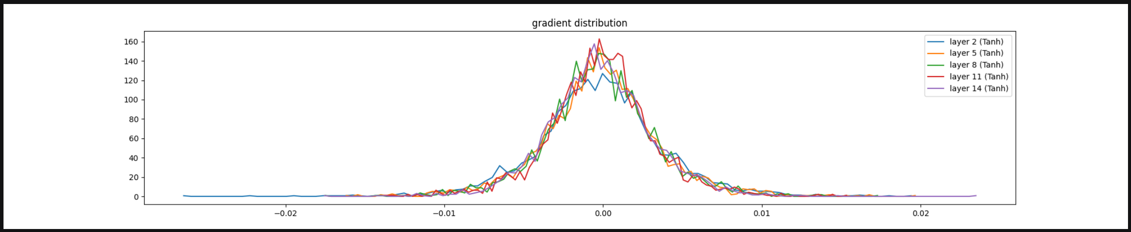
Diagnostic Tools and Visualization

1. Activation saturation of non-linear layers, i.e. tanh or ReLu etc. For tanh layers, want to make sure the distribution is fairly even, and not tailed towards ± 1 like in:



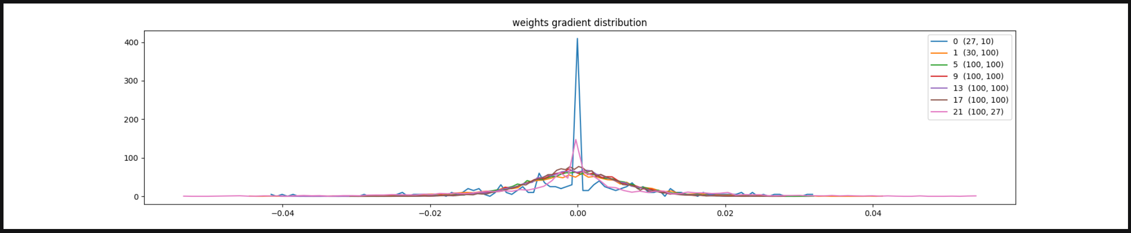
   Produced with 1. This is important, because neurons with activations of ± 1 will have gradients of 0, as $\frac{d}{dx}\tanh x = 1 - (\tanh x)^2$, leading to dead neurons which don't learn. These are forward pass statistics

2. Gradients of the non-linear layers. For tanh they can look like:



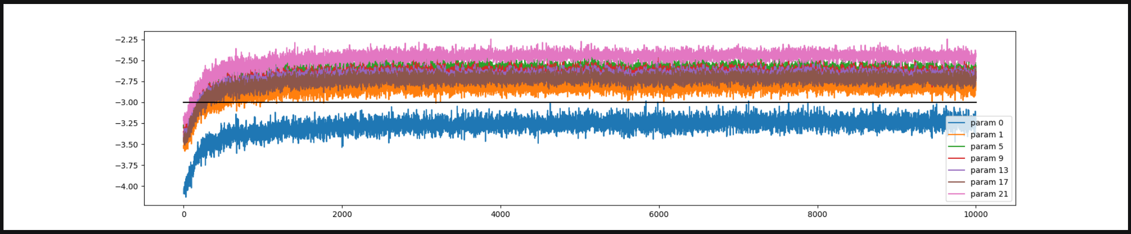
   Produced with 2. These are backward pass statistics since they involve the gradient

3. The ratio between the gradient and the parameter. Should be similar for each layer after the first and before the last. Restricting to 2-dim parameters, i.e. weights of the linear layers, not biases or γ and β in batch norm.



   Produced with 3.

4. The size of the updates relative to the parameters. As an estimate, want the $\log_{10}$ to be around 10^{-3}. This can be more informative than the gradient to data ratio, since what matters in learning is the size of the update.



Produced with 4.

Question

Why do we look at the ratio of grad standard deviations to parameter standard deviation (similarly for updates), instead of absolute values.

Batch Normalization

Batch normalization is a technique used to stabilize training. It computes the mean and sd of the batch and uses that to normalize the batch to a standard normal distribution. This helps ensure neuron activation is more stable. For example, if the non-linear layer is tanh, it will take values of ± 1 for inputs that are too far away from 0. Oversaturation could keep some neurons always on or off. It also means that adding a bias in a linear layer that comes before a batch normalization step is redundant as that bias is added to the norm and immediately subtracted, rendering it useless.

Backpropogation

Makemore Code Snippets

```
# visualize histograms of layer activation
plt.figure(figsize=(20, 4))
legends = []

for i, layer in enumerate(layers[:-1]): #exclude output layer
    if isinstance(layer, Tanh):
        t = layer.out
        print(’layer %d (%10s): mean %+.2f, std %.2f, saturated: %.2f%%’ % (i,
                layer.__class__.__name__,
                t.mean(), t.std(),
                (t.abs()>0.97).float().mean()*100))
        hy, hx = torch.histogram(t, density=True)
        plt.plot(hx[:-1].detach(), hy.detach())
        legends.append(f’layer {i} ({layer.__class__.__name__})’)
plt.legend(legends)
plt.title(’activation distribution’)
# plt.show()
```

Listing 1: Activation graph with matplotlib

```
    # histogram of gradients
plt.figure(figsize=(20, 4))
legends = []
for i, layer in enumerate(layers[:-1]): #exclude output layer
    if isinstance(layer, Tanh):
        t = layer.out.grad
        print(’layer %d (%10s): mean %+.2f, std %e’ % (i,
                layer.__class__.__name__,
                t.mean(), t.std()))
        hy, hx = torch.histogram(t, density=True)
        plt.plot(hx[:-1].detach(), hy.detach())
        legends.append(f’layer {i} ({layer.__class__.__name__})’)
plt.legend(legends)
```

```python
plt.title('gradient distribution')
```

Listing 2: Gradients of tanh layers with matplotlib

```python
# parameter values visualization
# scale of gradient compared to actual values
plt.figure(figsize=(20, 4))
legends = []
for i, p in enumerate(parameters): #exclude output layer
    t = p.grad
    if p.ndim == 2:
        print('weight %10s | mean %+f | std %e | grad:data ration %e' % (tuple(p.shape),
                t.mean(), t.std(), t.std() / p.std() ))
        hy, hx = torch.histogram(t, density=True)
        plt.plot(hx[:-1].detach(), hy.detach())
        legends.append(f'{i}  {tuple(p.shape)}')
plt.legend(legends)
plt.title('weights gradient distribution')
```

Listing 3: Scale between gradients and values of parameters

```python
#before training code
ud=[]
##### TRAINING #####
# in training loop
# lr is learning rate
# this tracks the log base 10
with torch.no_grad():
        ud.append([(lr*p.grad.std() / p.data.std()).log10().item() for p in parameters])
###################
# at each update, keep track of log of update size compated to data size
# these should be low or else we are over updating
# last layer can be large since that layer was ariticially compressed
plt.figure(figsize=(20,4))
legends = []
for i,p in enumerate(parameters):
    if p.ndim == 2:
        plt.plot([ud[j][i] for j in range(len(ud))])
        legends.append('param %d' % i)
plt.plot([0, len(ud)], [-3, -3], 'k') # ratios should be 1e-3
plt.legend(legends)
plt.show()
```

Listing 4: Size of update scaled by size of the parameter