```
    execve(args[0], args, environ);
}
```

```
*CTF{simple_simpler_simplest_stack_overflow}
```

misc/Alice's warm up

We are provided with a `data.pkl`, 6 serialized tensors in the `data` folder and a `hint.py` with the following contents:

```
import string
assert len(flag)==16
assert flagset=string.printable[0:36]+"*CTF{ALIZE}"
```

Looking at the `data.pkl`, we can deduce that the model is equivalent to three linear layers:

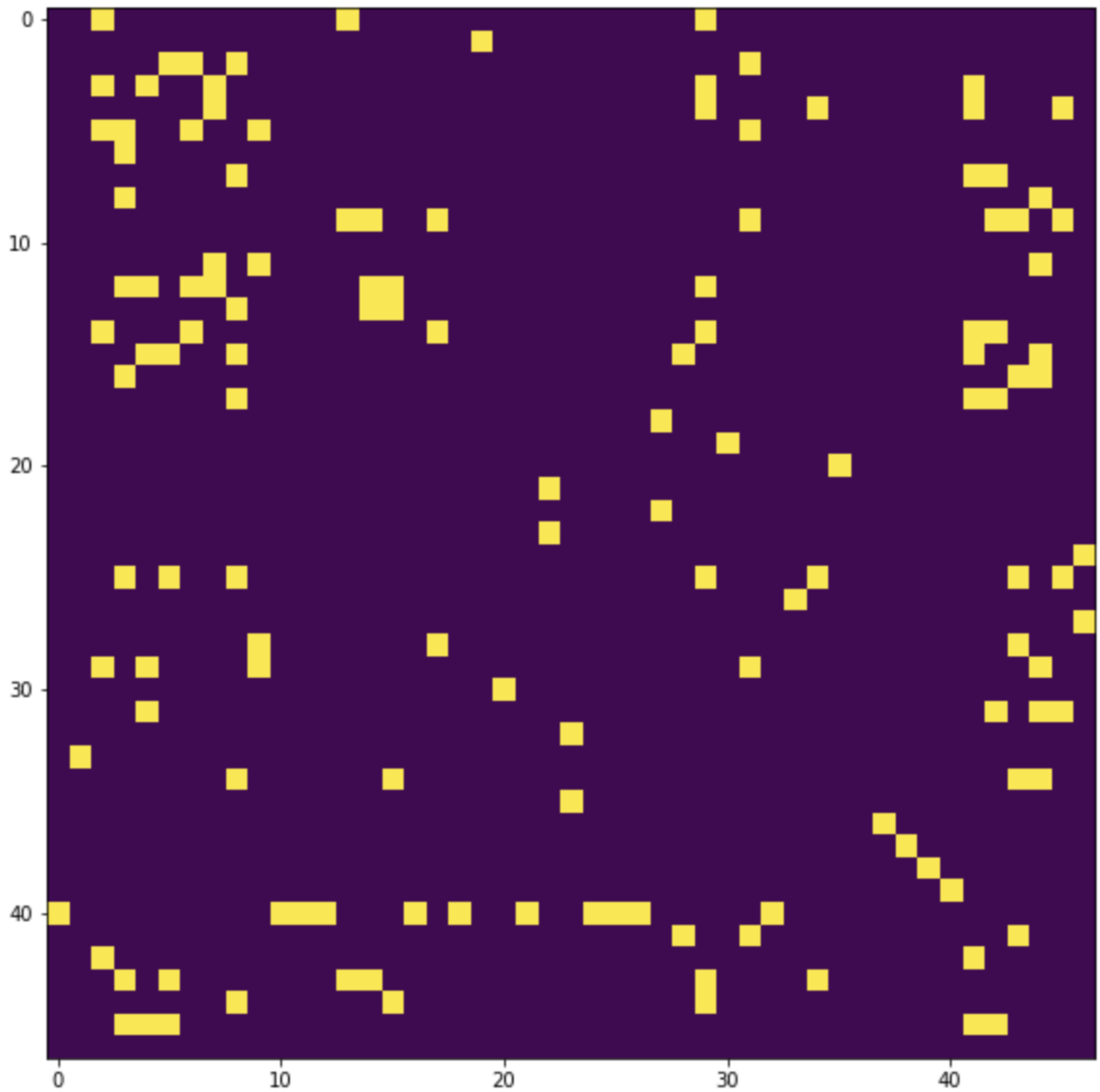
```
v0 = torch.tensor(np.frombuffer(open('./data/0', 'rb').read(), dtype=np.float32).re
v1 = torch.tensor(np.frombuffer(open('./data/1', 'rb').read(), dtype=np.float32).re
v2 = torch.tensor(np.frombuffer(open('./data/2', 'rb').read(), dtype=np.float32).re
v3 = torch.tensor(np.frombuffer(open('./data/3', 'rb').read(), dtype=np.float32).re
v4 = torch.tensor(np.frombuffer(open('./data/4', 'rb').read(), dtype=np.float32).re
v5 = torch.tensor(np.frombuffer(open('./data/5', 'rb').read(), dtype=np.float32).re

def forward(x):
    z0 = (v0 @ x) + v1
    z1 = (v2 @ z0) + v3
    z2 = (v4 @ z1) + v5
    x = torch.sigmoid(z2)
    return x
```

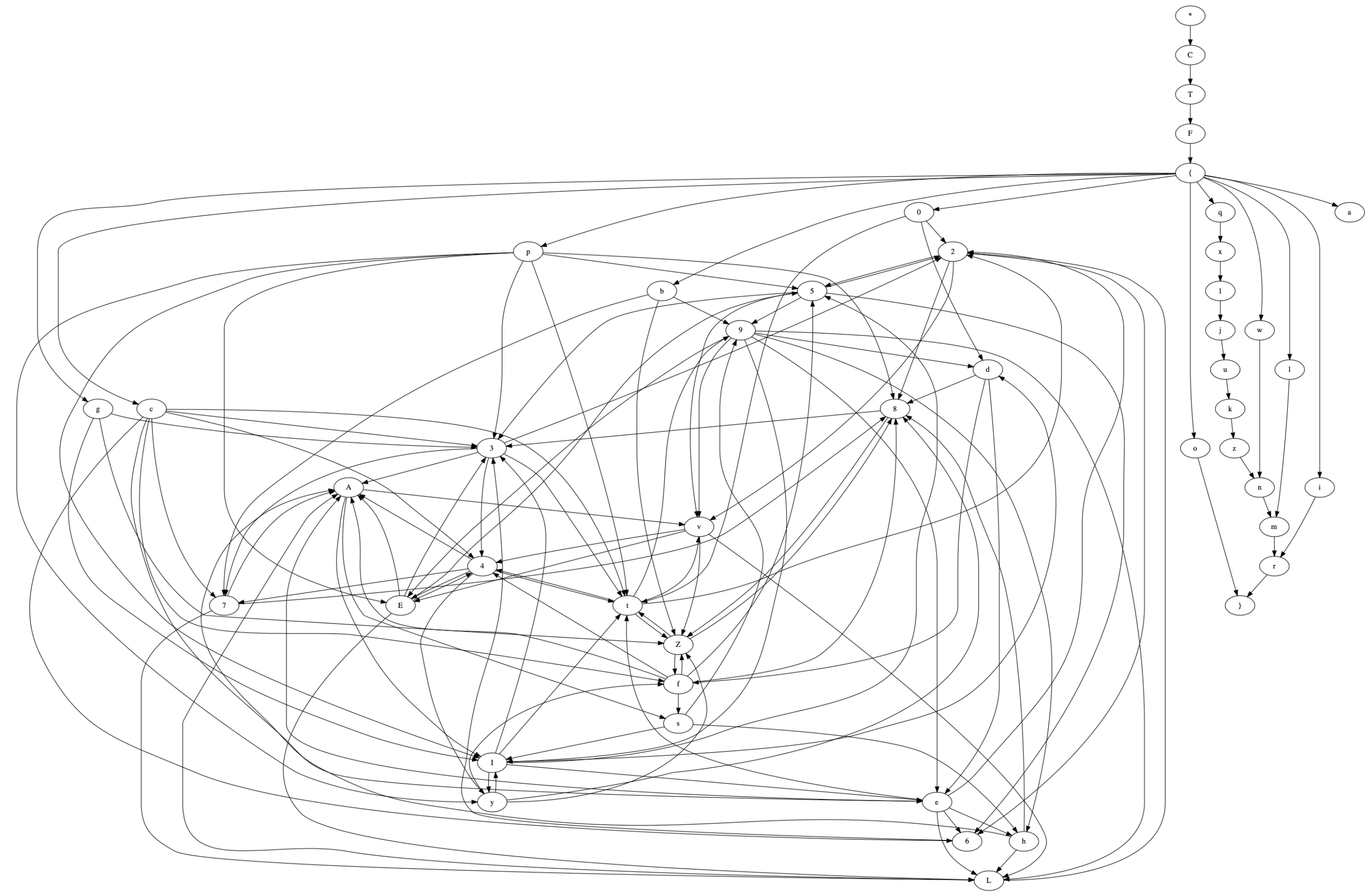
Effectively, this model maps a 47-size input vector to a 1-size output vector. Coincidentally, our available alphabet has 47 characters and therefore we can likely deduce that our input should be one-hot encoded.

However, given an input matrix of size `[47,16]` like we might expect, each individual character will be processed separately and therefore, this network couldn't implement flag checking.

We notice that one of the weight matrices, v0, contains only 0's and 1's despite being a float vector:



Furthermore, we notice an interesting pattern near the part of the alphebet with `*CTF` that indicates this might represent the transition function of a DFA. Upon plotting the transitions, we obtain:



On the right hand side, we see a clear path from `*CTF` to `}` that matches our expected size of 16: `*CTF{qx1jukznmr}`

misc/Alice’s challenge

In this challenge we are provied with `Net.model`, a serialized pytorch model and 25 gradient tensors in `grad.rar`.

Looking at the model, we see three convolutional layers followed by a linear layer. Although it is not present in the serialized model, we can deduce that there is a flattening step between the two and therefore we can reconstruct the model source:

```
class AliceNet2(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = Sequential(
            Conv2d(3, 12, kernel_size=(5,5), stride=(2,2), padding=(2,2)),
            Sigmoid(),
            Conv2d(12, 12, kernel_size=(5,5), stride=(2,2), padding=(2,2)),
            Sigmoid(),
            Conv2d(12, 12, kernel_size=(5,5), stride=(1,1), padding=(2,2)),
            Sigmoid(),
            Conv2d(12, 12, kernel_size=(5,5), stride=(1,1), padding=(2,2)),
            Sigmoid(),
        )
        self.fc = Sequential(
            Linear(768, 200)
        )

    def forward(self, x):
        z0 = self.conv(x)
        z1 = torch.reshape(z0, (-1,))
        z2 = self.fc(z1)
        return z2
```

With gradient tensors, our objective is likely to recover the original training samples that produced such values.

I.e. given F_θ and gradient Δ_k find x_k, y_k such that $L = ||F_\theta(x) - y||_2$ and $\frac{\partial L}{\partial \theta} = \Delta_k$. In order to recover x_k and y_k, we can create a new loss function: $L' = ||\frac{\partial L}{\partial \theta} - \Delta_k||_2$ and perform standard gradient descent on the input pair (x_k, y_k).

This concept is implemented in the following pytorch code:

```
model = torch.load('./Net.model', map_location=torch.device('cpu'))

tensors = [
    torch.load('./grad/%d.tensor' % x, map_location=torch.device('cpu'))
    for x in range(25)
]

def mse(a,b):
    return torch.sum((a-b)**2)

def compute_grad_loss(x0,y0,t):
    y1 = model(x0)

    loss = torch.mean((y1 - y0) ** 2)
    loss.backward(retain_graph=True)

    grad_loss = 0
    for a,b in zip(t, model.parameters()):
        b0 = torch.autograd.grad(loss, b, retain_graph=True, create_graph=True)[0]
        grad_loss += mse(a,b0)

    # Compute gradient w.r.t our input pair (x,y)
    dx = torch.autograd.grad(grad_loss, x0, retain_graph=True)[0]
    dy = torch.autograd.grad(grad_loss, y0, retain_graph=True)[0]

    return dx, dy, grad_loss

def solve_image(t):
    x = torch.zeros(1,3,32,32)
    y = torch.zeros(1,200)

    alpha = 0.1

    for i in tqdm(range(500)):
```

```
        x0 = x.clone()
        y0 = y.clone()
        x0.requires_grad = True
        y0.requires_grad = True

        dx, dy, grad_loss = compute_grad_loss(x0, y0, t)

        x -= dx * alpha
        y -= dy * alpha

    return x

x = [solve_image(t) for t in tensors]
```

We obtain pretty clear original images after only 500 steps:

```
*CTF{PZEXHBEARAK8MQT5NAliceE}
```

misc/babyFL