Computation Graphs

- Computing gradients is hard!
- Automatic differentiation: instrument code to keep track of derivatives

```
y = x * x   --codegen-->   (y,dy) = (x * x, 2 * x * dx)
```

- Computation is now something we need to reason about symbolically; use a library like PyTorch (or Tensorflow)

PyTorch

- Framework for defining computations that provides easy access to derivatives

- Module: defines a neural network (can use wrap other modules which implement predefined layers)

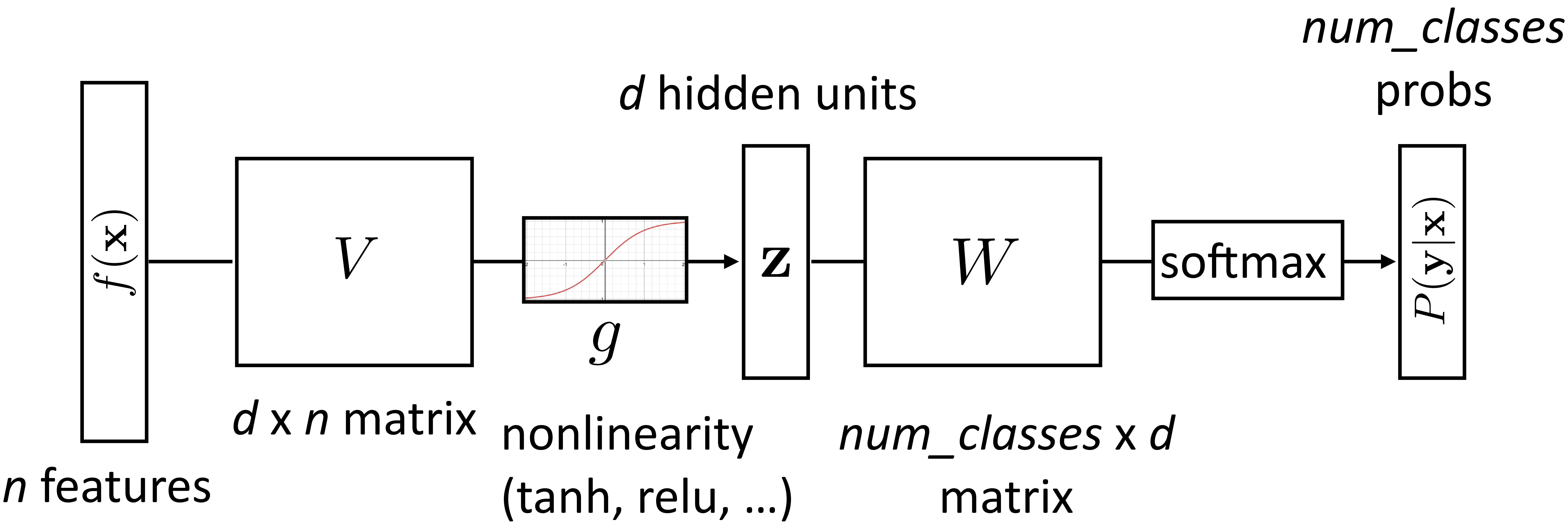
- If forward() uses crazy math, you have to write backward yourself

```
torch.nn.Module

    # Takes an example x and computes result
    forward(x):
        …
    # Computes gradient after forward() is called
    backward(): # produced automatically
        …
```

Neural Networks for Classification

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(W g(V f(\mathbf{x})))$$



Computation Graphs in PyTorch

- Define forward pass for $P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$

```
class FFNN(nn.Module):
    def __init__(self, input_size, hidden_size, out_size):
        super(FFNN, self).__init__()
        self.V = nn.Linear(input_size, hidden_size)
        self.g = nn.Tanh() # or nn.ReLU(), sigmoid()...
        self.W = nn.Linear(hidden_size, out_size)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

(syntactic sugar for forward)

Input to Network

- Whatever you define with torch.nn needs its input as some sort of tensor, whether it's integer word indices or real-valued vectors

```
def form_input(x) -> torch.Tensor:
  # Index words/embed words/etc.
  return torch.from_numpy(x).float()
```

- torch.Tensor is a different datastructure from a numpy array, but you can translate back and forth fairly easily

- Note that **translating out of PyTorch will break backpropagation**; don't do this inside your Module

Training and Optimization

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$

one-hot vector
of the label
(e.g., [0, 1, 0])

```
ffnn = FFNN(inp, hid, out)
optimizer = optim.Adam(ffnn.parameters(), lr=lr)
for epoch in range(0, num_epochs):
    for (input, gold_label) in training_data:
        ffnn.zero_grad() # clear gradient variables
        probs = ffnn.forward(input)
        loss = torch.neg(torch.log(probs)).dot(gold_label)
        loss.backward()
        optimizer.step()
```

negative log-likelihood of correct answer

Computing Gradients with Backprop

```
class FFNN(nn.Module):
    def __init__(self, inp, hid, out):
        super(FFNN, self).__init__()
        self.V = nn.Linear(inp, hid)
        self.g = nn.Tanh()
        self.W = nn.Linear(hid, out)
        self.softmax = nn.Softmax(dim=0)
        nn.init.uniform(self.V.weight)
```

- Initializing to a nonzero value is critical!

Training a Model

Define modules, etc.

Initialize weights and optimizer

For each epoch:

    For each batch of data:

        Zero out gradient

        Compute loss on batch

        Autograd to compute gradients and take step on optimizer

    [Optional: check performance on dev set to identify overfitting]

Run on dev/test set