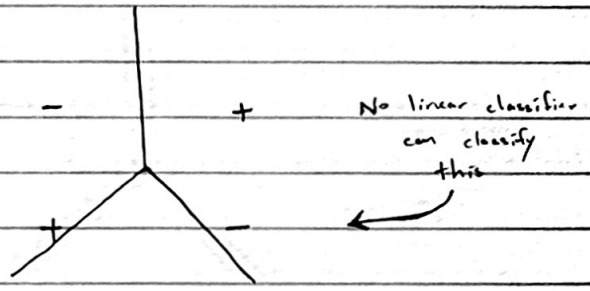
Neural Networks

	$f(\bar{x})$ G	B	N
good	1	0	0
bad	0	1	0
not good	1	0	1
not bad	0	1	1



- Neural Nets : Transform data into latent feature space

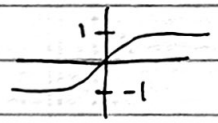
$\bar{w}^T \underbrace{f(\bar{x})}$ ⟶ replace $f(\bar{x})$ with a non-linear function of the original $f(\bar{x})$

Define $\bar{z} = g\left(\underbrace{V}_{d \times n \text{ matrix}} \underbrace{f(\bar{x})}_{n\text{-dimensional feature vector}}\right)$

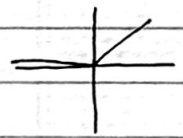
non-linearity

Types of non-linearity

$g = \tanh$



$g = \text{ReLU}$

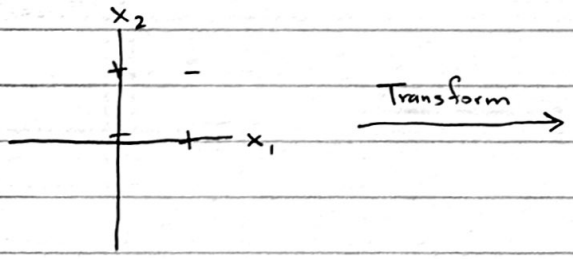


How can $V + g$ give us useful latent features?

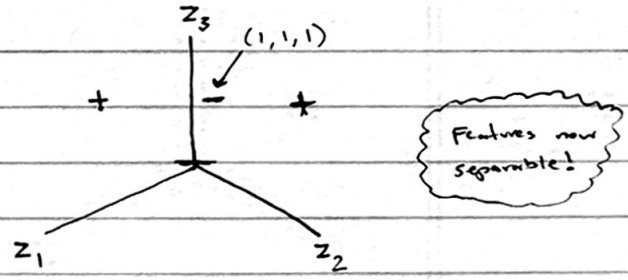
Ex. Suppose $V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$ $g = \tanh$

$\tanh(0) = 0$, $\tanh(1) \approx 1$, $\tanh(2) \approx 1$

$$\bar{z} = g\left(V \underbrace{f(\bar{x})}_{(x_1, x_2)}\right) = \left[\tanh(x_1), \tanh(x_2), \tanh(x_1 + x_2)\right]$$



OG Feature Space



New Feature Space

Feed Forward Neural Networks

- Vectorization and Softmax

Multiclass Logistic Regression "Diff. weights"

$$P(y|x) = \frac{\exp(w_y^T f(x))}{\sum_{y' \in y} \exp(w_{y'}^T f(x))}$$

* Single scalar probability

3 classes "Different Weights"

$$w_1^T = -1.1 \quad w_2^T = 2.1 \quad w_3^T = -0.4$$

softmax →

$$0.036 \quad 0.89 \quad 0.07$$

class probabilities

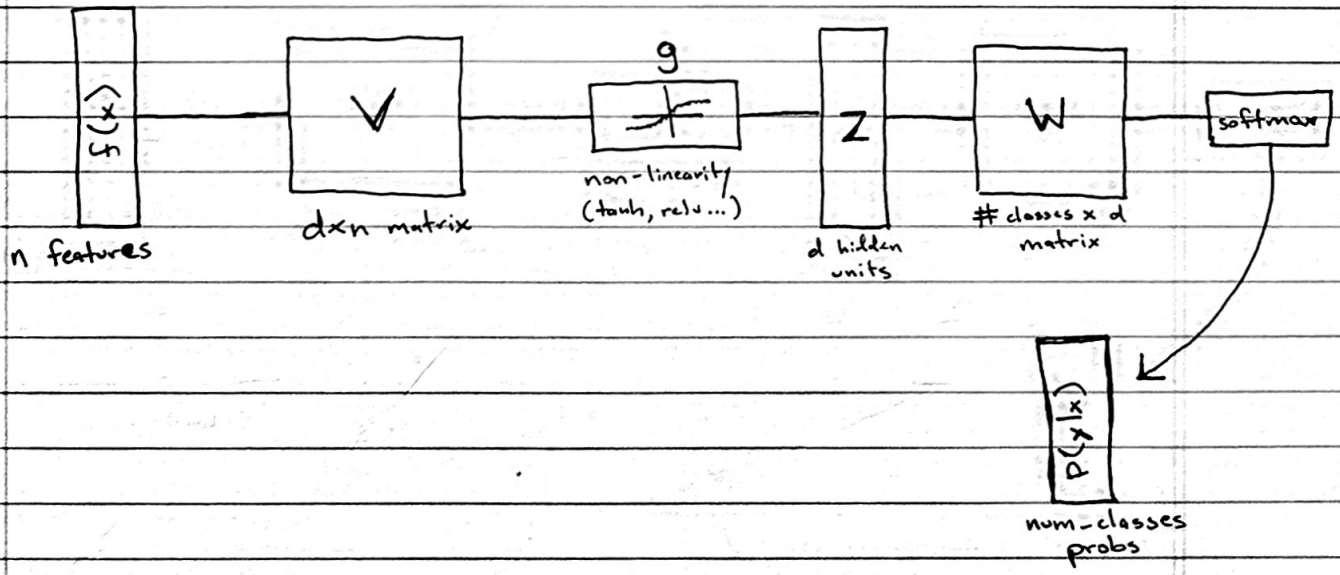
- Softmax operation: exponentiate and normalize
  - Divide by their sum
- Written as: $\text{softmax}(W f(x))$

$$P(y|x) = \text{softmax}(W f(x))$$

* Weight vector per class
W is [# classes x # feats]

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

* Now one hidden layer



- **Training Neural Nets**

stage 2 stage 1

$$P(y|x) = \text{softmax}(Wz) \qquad z = g(Vf(x))$$

- Maximize log likelihood of training data:

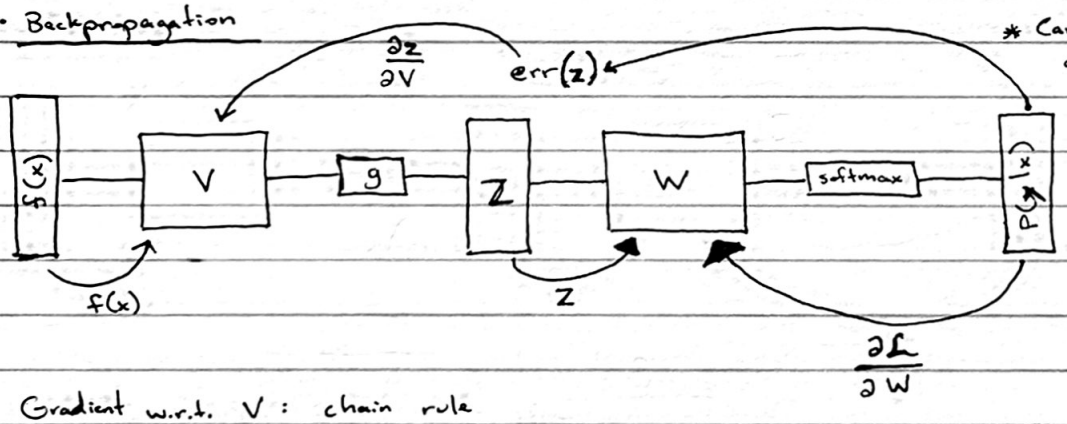
$$\mathcal{L}(x, i^*) = \log P(y = i^*|x) = \log(\text{softmax}(Wz) \cdot e_{i^*})$$

i^* = index of gold label

e_i = 1 in the i^{th} row, 0 elsewhere. Dot by this = select i^{th} index

$$\mathcal{L}(x, i^*) = Wz \cdot e_{i^*} - \log \sum_j \exp(Wz) \cdot e_j$$

- **Backpropagation**



* Can ignore everything after z, treat as output and keep backpropping

Gradient w.r.t. V: chain rule

$$\frac{\partial \mathcal{L}(x, i^*)}{\partial V_{ij}} = \underbrace{\frac{\partial \mathcal{L}(x, i^*)}{\partial z}}_{1} \cdot \underbrace{\frac{\partial z}{\partial V_{ij}}}_{2}$$

* Gradient w.r.t. W: looks like logistic regression, can be computed using z as the features

First Term (1): err(z) represents gradient w.r.t. z

Second Term (2):

$$\frac{\partial z}{\partial V_{ij}} = \underbrace{\frac{\partial g(a)}{\partial a}}_{\text{gradient of non-linear activation function at } a \text{ (depends on current value)}} \cdot \underbrace{\frac{\partial a}{\partial V_{ij}}}_{\text{gradient of linear function}}$$

Implementing Neural Nets

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

$$y = x \cdot x \xrightarrow[\text{gen}]{\text{code}} (y, dy) = (x \cdot x,\ 2 \cdot x \cdot dx)$$

- PyTorch: Framework for defining computations that provides easy access to derivatives

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

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

```
torch.nn.module:

  # Takes example x and computes result
    forward(x):
      ...

  # Computes gradient after forward called
    backward():
      ... produced automatically!
```

Code and Pseudocode on "seg-15.pdf"

- Batching: Batching data gives speedups due to more efficient matrix operations
  - Batch sizes from 1-100 often work well

- How does initialization affect learning?
  - How do we initialize V and W?
  - Non-convex problem. Initialization matters!
  - Must initialize to non-zero to get model to learn
  - If you initialize too large, cells become saturated, gradient close to 0, no learning
  - Can do random uniform/normal initialization w/ appropriate scale

fan-in: # inputs
fan-out: # outputs

$$\text{Glorot Initializer}: U\left[-\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}},\ +\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}\right]$$

* Want variance and gradient of inputs for each layer to be the same

- **Dropout**: Probabilistically zero out parts of network during training to prevent overfitting. Use whole network at test time
  - Form of stochastic regularization
  - Similar to benefits of ensembling (sub-networks)
  - One line of code

- **Gradient Clipping**: Set max value for gradients