Neural Nets

NNs: transform the data into a latent feature space

$\bar{w}^T f(\bar{x})$ replace $f(\bar{x})$ with a nonlinear func of the orig $f(\bar{x})$

Define $\bar{z} = g(V f(\bar{x}))$

$V f(\bar{x})$: n-dimensional feat vector

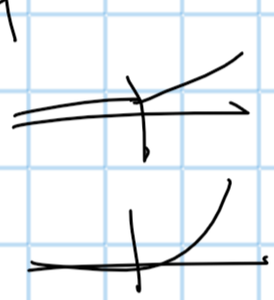
V: d×n matrix

g: nonlinearity

How can Vf, g give us useful latent features?

classify with $\bar{w}^T \bar{z}(\bar{x})$

$g = \tanh$ 

$g = \text{ReLU}$ 

---

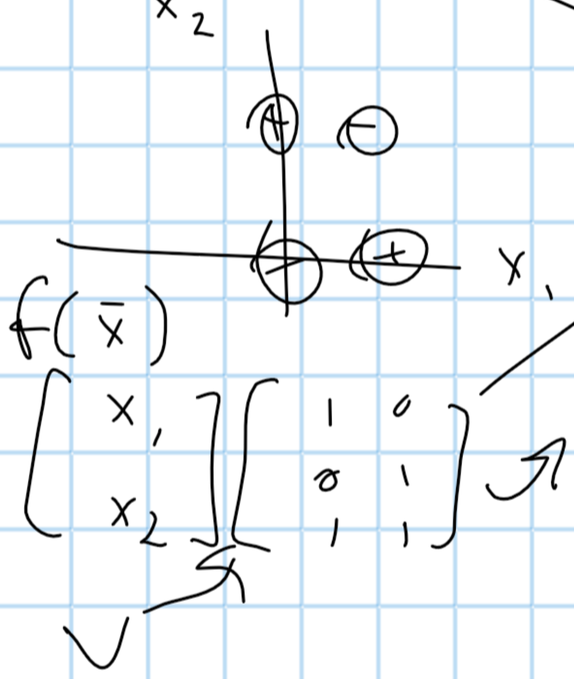
ex:

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

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

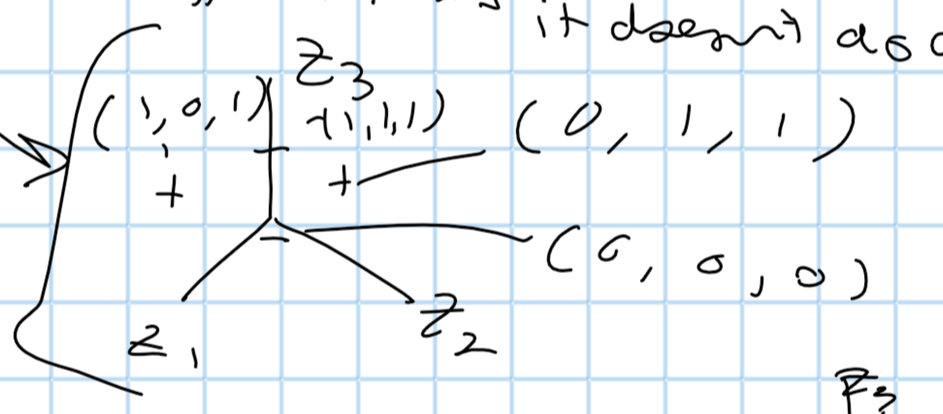
(increasing to 1 "activates" but after that it doesn't do anything

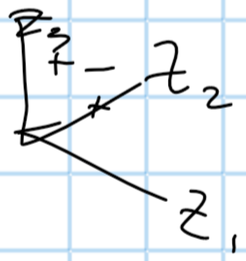
$z = [\tanh(x_1), \tanh(x_2), \tanh(x_1 + x_2)]$



$\bar{z} = g(V f(\bar{x}))$

$f(\bar{x}) = (x_1, x_2)$





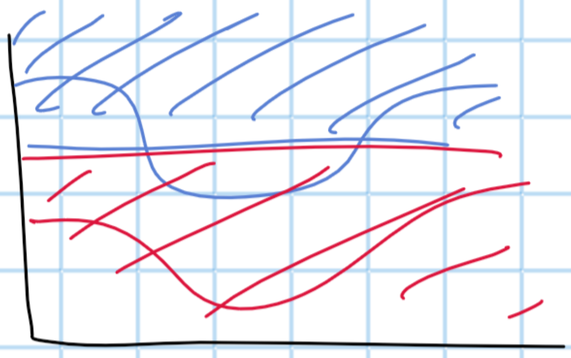
Neural Network Visualization

nonlinear warp shift

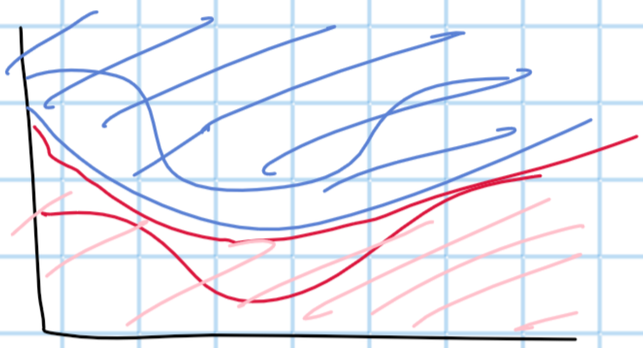
$z = g(V f(x) + b)$ we can ignore $+b$

$y_{pred} = \arg\max_y W_y^T z$

Linear classifier



NN



Deep Neural Networks — multiple layers

learn transformation with backpropagation

$z_1 = g(V_1 f(x))$

$z_2 = g(V_2 z_1)$

...

$y_{pred} = \arg\max_y W_y^T z_n$

Feedforward Neural Networks, Backpropagation

LR: $P(y|x) = \frac{\exp(w_y^T f(x))}{\sum_{y' \in Y} \exp(w_{y'}^T f(x))}$ single scalar probability

3 classes "diff weights"

$w_1^T f(x)$	-1.1	softmax →	0.036
$w_2^T f(x)$	2.1		0.89
$w_3^T f(x)$	-0.4		0.07

class probs

Softmax operation = "exp and normalize"

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

weight vector per class

W is [num classes x num feats]

$P(y|x) = \text{softmax}(W_g(V f(x)))$ 1 hidden layer

num classes

$f(x)$ → $\cdot V$ (d x n mat) → g (nonlinearity) → z (d hidden units) → $\cdot W$ (num classes x d mat) → softmax → $P(y|x)$ (probs)

n features

Training

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

Maximize log likelihood of training data

i^*: index of the gold label

e_i: 1 in the i^{th} row, zero elsewhere (num classes x 1 vector)

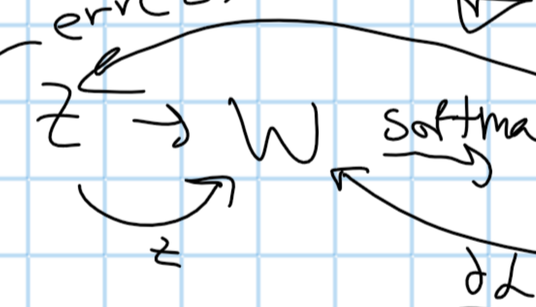
$L(x, i^*) = \log P(y = i^*|x) = \log(\text{softmax}(Wz) \cdot e_{i^*})$

in a 3 class problem, if the i^* in the 2nd pos, use e_2

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

dotting by this

Can forget everyth after z, treat it as the output and keep bp



∇ wrt W: looks like LR func, can be computed treating z as the features

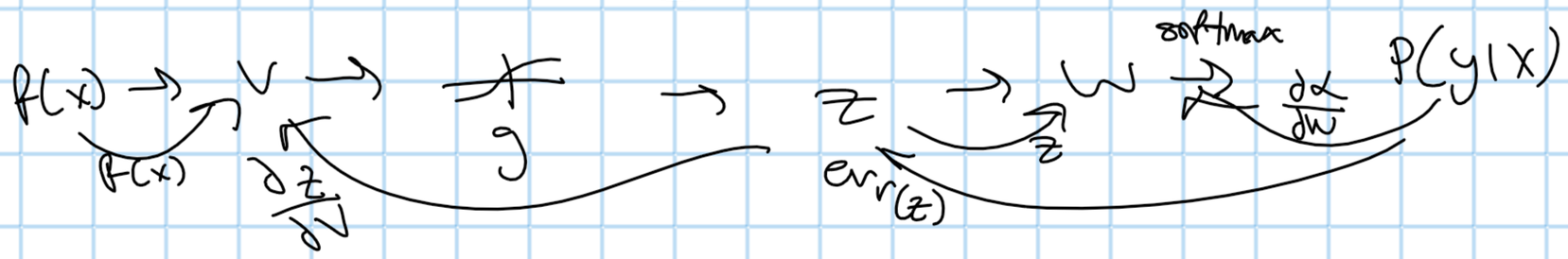
Computing Gradients with backprop:

$L(x, i^*) = Wz \cdot e_{i^*} - \log \sum \exp(Wz) \cdot e_j \quad z = g(Vf(x))$ activations at hidden layer

gradient wrt V: apply chain rule

$$\frac{\partial L(x, i^*)}{\partial V_{ij}} = \frac{\partial L(x, i^*)}{\partial z} \cdot \frac{\partial z}{\partial V_{ij}} \qquad \frac{\partial z}{\partial V_{ij}} = \frac{\partial g(a)}{\partial a} \cdot \frac{\partial a}{\partial V_{ij}} \qquad a = V f(x)$$

($\frac{\partial L}{\partial z}$: error)



Neural Network Implementation

automatic differentiation

$y = x * x \rightarrow (y, dy) = (x * x, 2 \cdot x \cdot dx)$

torch.nn.Module

takes an example x and computes result
forward(x).

Computes gradient after forward is called
backward() is produced automatically

Training and optimization

```
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)

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

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

Training:

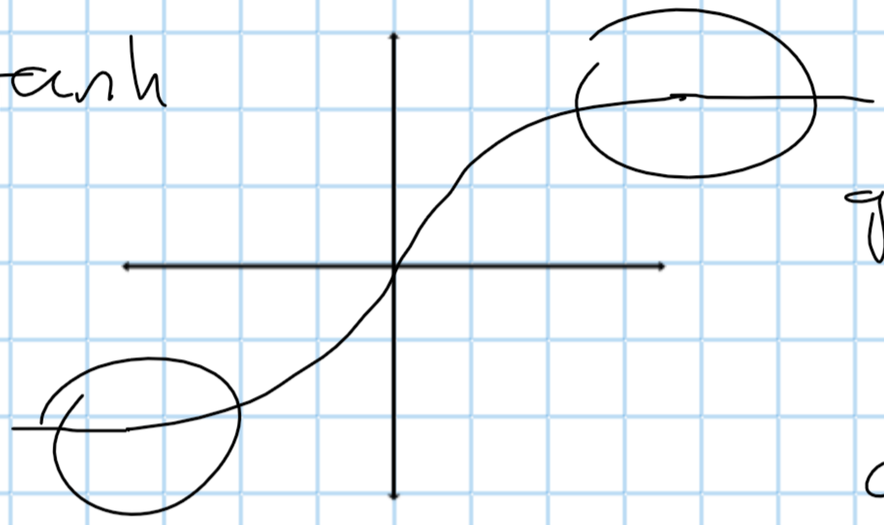
Define modules, etc.
init weights and optimizer
For each epoch
    for each batch of data
        zero gradient
        compute loss on batch
        autograd to compute gradients and take step on optimizer
    (Optional) check performance on dev set so no overfitting
Run on dev/test set

Neural Net Training, optimization

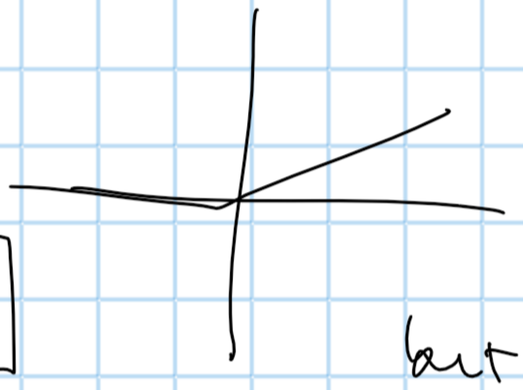
Initialization

nonconvex problem, so init matters (ex: 0 for V will not work)

tanh



gradients are small here, so model thinks that there is little change. So if inputs are too big (+ or -), the model won't change. So to avoid, you can use ReLU



can do random initialization with appropriate scale

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

want variance of inputs + gradients to be the same

but if many neg nums, same problem can happen

batch normalization is rescaling each layer to have mean 0 and var 1 over a batch
this is useful if the net is deep (more applicable in CV than NLP) (Ioffe & Szegedy, 2015)

Dropout

zero out some neurons during training to prevent overfitting

too much dropout leads to bad performance, good dropout has some benefit

Optimizer

Adam (Kingma and Ba, ICLR2015): adaptive step + momentum

gradient clipping sets max value for gradients