Deep networks

CS 446
These lectures will follow an ERM perspective on deep networks:

- Pick a **model/predictor class (network architecture)**.
  (We will spend most of our time on this!)

- Pick a **loss/risk**.
  (We will almost always use cross-entropy!)

- Pick an **optimizer**.
  (We will mostly treat this as a black box!)

The goal is low **test error**, whereas above only gives low **training error**; we will briefly discuss this as well.
1. Linear networks.
Iterated linear predictors

The most basic view of a neural network is an iterated linear predictor.

- 1 layer: 
  \[ x \mapsto W_1 x + b_1. \]

- 2 layers: 
  \[ x \mapsto W_2 (W_1 x + b_1) + b_2. \]

- 3 layers: 
  \[ x \mapsto W_3 (W_2 (W_1 x + b_1) + b_2) + b_3. \]

- \( L \) layers: 
  \[ x \mapsto W_L (\cdots (W_1 x + b_1) \cdots) + b_L. \]

Alternatively, this is a composition of linear predictors:

\[ x \mapsto (f_L \circ f_{L-1} \circ \cdots \circ f_1) (x), \]

where \( f_i(z) = W_i z + b_i \) is an affine function.

Note: “layer” terminology is ambiguous, we’ll revisit it.
Wait a minute...

Note that

\[ W_L (\cdots (W_1 x + b_1) \cdots) + b_L \]
\[ = (W_L \cdots W_1) x + (b_L + W_L b_{L-1} + \cdots + W_L \cdots W_2 b_1) \]
\[ = w^T [x_1], \]

where \( w \in \mathbb{R}^{d+1} \) is

\[ w_{1:d}^T = W_L \cdots W_1, \quad w_{d+1} = b_L + W_L b_{L-1} + \cdots + W_L \cdots W_2 b_1. \]

Oops, this is just a linear predictor.
2. Activations/nonlinearities.
Recall that logistic regression could be interpreted as a probability model:

\[
\Pr[Y = 1 | X = x] = \frac{1}{1 + \exp(-\mathbf{w}^\top \mathbf{x})} =: \sigma_s(\mathbf{w}^\top \mathbf{x}),
\]

where \( \sigma_s \) is the logistic or sigmoid function.

Don't worry, we'll slow down next slide; for now, iterated logistic regression gave our first deep network!

Remark: can view intermediate layers as features to subsequent layers.
Iterated **logistic regression**

Recall that logistic regression could be interpreted as a probability model:

$$\Pr[Y = 1 | X = x] = \frac{1}{1 + \exp(-w^T x)} =: \sigma_s(w^T x),$$

where $\sigma_s$ is the **logistic** or **sigmoid** function.

Now suppose $\sigma_s$ is applied coordinate-wise, and consider

$$x \mapsto (f_L \circ \cdots \circ f_1)(x) \quad \text{where} \quad f_i(z) = \sigma_s(W_i z + b_i).$$

Don't worry, we'll slow down next slide; for now, iterated logistic regression gave our first deep network!

Remark: can view intermediate layers as **features** to subsequent layers.
Basic deep networks

A self-contained expression is

$$x \mapsto \sigma_L \left( W_L \sigma_{L-1} \left( \cdots \left( W_2 \sigma_1 (W_1 x + b_1) + b_2 \right) \cdots \right) + b_L \right),$$

with equivalent “functional form”

$$x \mapsto (f_L \circ \cdots \circ f_1)(x) \quad \text{where } f_i(z) = \sigma_i (W_i z + b_i).$$

Some further details (many more to come!):

- $(W_i)^L_{i=1}$ with $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$ are the weights, and $(b_i)^L_{i=1}$ are the biases.
- $(\sigma_i)^L_{i=1}$ with $\sigma_i : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_i}$ are called nonlinearities, or activations, or transfer functions, or link functions.
- This is only the basic setup; many things can and will change, please ask many questions!
Choices of activation

Basic form:

\[ x \mapsto \sigma_L \left( W_L \sigma_{L-1} ( \cdots W_2 \sigma_1 (W_1 x + b_1) + b_2 \cdots ) + b_L \right). \]

Choices of activation (univariate, coordinate-wise):

- **Indicator/step/heavyside/threshold** \( z \mapsto 1[z \geq 0] \).
  This was the original choice (1940s!).

- **Sigmoid** \( \sigma_s(z) := \frac{1}{1+\exp(-z)} \).
  This was popular roughly 1970s - 2005?

- **Hyperbolic tangent** \( z \mapsto \tanh(z) \).
  Similar to sigmoid, used during same interval.

- **Rectified Linear Unit (ReLU)** \( \sigma_r(z) = \max\{0, z\} \).
  It (and slight variants, e.g., Leaky ReLU, ELU, ...) are the dominant choice now; popularized in “Imagenet/AlexNet” paper (Krizhevsky-Sutskever-Hinton, 2012).

- **Identity** \( z \mapsto z \); we’ll often use this as the last layer when we use cross-entropy loss.

- **NON-coordinate-wise choices**: we will discuss “softmax” and “pooling” a bit later.
“Architectures” and “models”

Basic form:

\[ x \mapsto \sigma_L \left( W_L \sigma_{L-1} \left( \cdots W_2 \sigma_1 (W_1 x + b_1) + b_2 \cdots \right) + b_L \right). \]

\(((W_i, b_i))_{i=1}^{L}, \text{ the weights and biases, are the parameters.} \)

Let’s roll them into \( \mathcal{W} := (((W_i, b_i))_{i=1}^{L}, \text{ and consider the network as a two-parameter function } F_{\mathcal{W}}(x) = F(x; \mathcal{W}). \)

- The model or class of functions is \( \{F_{\mathcal{W}} : \text{all possible } \mathcal{W}\}. \) \( F \) (both arguments unset) is also called an architecture.
- When we fit/train/optimize, typically we leave the architecture fixed and vary \( \mathcal{W} \) to minimize risk.

(More on this in a moment.)
ERM recipe for basic networks

Standard ERM recipe:

▶ First we pick a class of functions/predictors; for deep networks, that means a $F(\cdot, \cdot)$.

▶ Then we pick a loss function and write down an empirical risk minimization problem; in these lectures we will pick cross-entropy:

$$\arg \min_{W} \frac{1}{n} \sum_{i=1}^{n} \ell_{ce}(y_i, F(x_i, W))$$

$$= \arg \min_{W_1 \in \mathbb{R}^{d \times d_1}, b_1 \in \mathbb{R}^{d_1}} \frac{1}{n} \sum_{i=1}^{n} \ell_{ce}(y_i, F(x_i; (W_i, b_i))_{i=1}^{L})$$

$$= \arg \min_{W_1 \in \mathbb{R}^{d \times d_1}, b_1 \in \mathbb{R}^{d_1}} \frac{1}{n} \sum_{i=1}^{n} \ell_{ce}(y_i, \sigma_L(\cdots \sigma_1(W_1 x_i + b_1) \cdots))$$

▶ Then we pick an optimizer. In this class, we only use gradient descent variants. It is a miracle that this works.
Remark on affine expansion

Note: we are writing

$$x \mapsto \sigma_L \left( \cdots \left( W_2 \sigma_1 \left( W_1 x + b_1 \right) + b_2 \right) \cdots \right),$$

rather than

$$x \mapsto \sigma_L \left( \cdots \left( W_2 \sigma_1 \left( W_1 \left[ x \right] \right) \right) \cdots \right).$$

▶ First form seems natural:
With “iterated linear prediction” perspective, it is natural to append 1 at every layer.

▶ Second form is sufficient:
with ReLU, $\sigma_r(1) = 1$, so can pass forward the constant; similar (but more complicated) options exist for other activations.

▶ Why do we do it?
It seems to make the optimization better behaved; this is currently not well understood.
Which architecture?

How do choose an architecture?

▶ How did we choose $k$ in $k$-nn?

Split data into training and validation, train different architectures and evaluate them on validation, choose architecture with lowest validation error.

▶ As with other methods, this is a proxy to minimizing test error.

Note. ▶ For many standard tasks (e.g., classification of standard vision datasets), people know good architectures.

▶ For new problems and new domains, things are absolutely not settled.
Which architecture?

How do choose an architecture?

▸ How did we choose $k$ in $k$-nn?
How do choose an architecture?

- How did we choose $k$ in $k$-nn?
- Split data into training and validation, train different architectures and evaluate them on validation, choose architecture with lowest validation error.
- As with other methods, this is a proxy to minimizing test error.
Which architecture?

How do choose an architecture?

▶ How did we choose $k$ in $k$-nn?

▶ Split data into training and validation, train different architectures and evaluate them on validation, choose architecture with lowest validation error.

▶ As with other methods, this is a proxy to minimizing test error.

Note.

▶ For many standard tasks (e.g., classification of standard vision datasets), people know good architectures.

▶ For new problems and new domains, things are absolutely not settled.
3. What we have gained: representation power
Sometimes, linear just isn’t enough

Linear predictor:
\[ x \mapsto w^T [x_1]. \]
Some blue points misclassified.

ReLU network:
\[ x \mapsto W_2 \sigma_r(W_1 x + b_1) + b_2. \]
0 misclassifications!
Classical example: XOR

Classical “XOR problem” (Minsky-Papert-'69).
(Check wikipedia for “AI Winter”.)

Theorem. On this data, any linear classifier (with affine expansion) makes at least one mistake.

Picture proof. Recall: linear classifiers correspond to separating hyperplanes.
Classical example: XOR

Classical “XOR problem” (Minsky-Papert-'69).
(Check wikipedia for “AI Winter”.)

**Theorem.** On this data, any linear classifier (with affine expansion) makes at least one mistake.

**Picture proof.** Recall: linear classifiers correspond to separating hyperplanes.

▶ If it splits the blue points, it’s incorrect on one of them.
Classical example: XOR

Classical “XOR problem” (Minsky-Papert-’69).
(Check wikipedia for “AI Winter”.)

Theorem. On this data, any linear classifier (with affine expansion) makes at least one mistake.

Picture proof. Recall: linear classifiers correspond to separating hyperplanes.

- If it splits the blue points, it’s incorrect on one of them.
- If it doesn’t split the blue points, then one halfspace contains the common midpoint, and therefore wrong on at least one red point.
One layer was not enough. How about two?

**Theorem** (Cybenko ’89, Hornik-Stinchcombe-White ’89, Funahashi ’89, Leshno et al ’92, . . .). Given any continuous function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ and any $\epsilon > 0$, there exist parameters $(W_1, b_1, W_2)$ so that

$$\sup_{\mathbf{x} \in [0,1]^d} \left| f(\mathbf{x}) - W_2 \sigma (W_1 \mathbf{x} + b_1) \right| \leq \epsilon,$$

as long as $\sigma$ is “reasonable” (e.g., ReLU or sigmoid or threshold).
One layer was not enough. How about two?

**Theorem** (Cybenko '89, Hornik-Stinchcombe-White '89, Funahashi '89, Leshno et al '92, ...). Given any continuous function \( f : \mathbb{R}^d \to \mathbb{R} \) and any \( \epsilon > 0 \), there exist parameters \((W_1, b_1, W_2)\) so that

\[
\sup_{x \in [0,1]^d} \left| f(x) - W_2 \sigma(W_1 x + b_1) \right| \leq \epsilon,
\]

as long as \( \sigma \) is “reasonable” (e.g., ReLU or sigmoid or threshold).

**Remarks.**

- Together with XOR example, justifies using nonlinearities.
- Does *not* justify (very) deep networks.
- Only says these networks *exist*, not that we can optimize for them!
4. Network/graph interpretation
Classical network/graph perspective

\[ v := \sigma(z), \quad z = \sum_{i=1}^{d} w_{i} x_{i}. \]
Classical network/graph perspective

\[ v_j := \sigma(z_j), \quad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \quad j \in \{1, 2\}. \]
Classical network/graph perspective

\[ v_j := \sigma(z_j), \quad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \quad j \in \{1, \ldots, k\}. \]
Columns of $\mathbf{W}_1 \in \mathbb{R}^{d \times k}$: params. of original logistic regression models.

Columns of $\mathbf{W}_2 \in \mathbb{R}^{k \times k}$: params. of new logistic regression models to combine predictions of original models.
Multilayer neural network

- Columns of $W_1 \in \mathbb{R}^{d \times k}$: params. of original logistic regression models.
- Columns of $W_2 \in \mathbb{R}^{k \times k}$: params. of new logistic regression models to combine predictions of original models.
- Non-input nodes ("units") compute $z \mapsto \sigma(w^T z + b)$ for some $(w, b)$. 

$\sigma(x) = \frac{1}{1 + e^{-x}}$
Multilayer neural network

- Columns of $W_1 \in \mathbb{R}^{d \times k}$: params. of original logistic regression models.
- Columns of $W_2 \in \mathbb{R}^{k \times k}$: params. of new logistic regression models to combine predictions of original models.

- Non-input nodes ("units") compute $z \mapsto \sigma(w^T z + b)$ for some $(w, b)$.
- Non-input and non-output units are called hidden.
General graph-based view

Classical graph-based perspective.

- Network is a directed acyclic graph;
  sources are inputs, sinks are outputs, intermediate nodes compute
  \[ z \mapsto \sigma(w^T z + b) \] (with their own \((\sigma, w, b)\)).

- Nodes at distance 1 from inputs are the first layer, distance 2 is second
  layer, and so on.

“Modern” graph-based perspective.

- Edges in the graph can be multivariate, meaning vectors or general
  tensors, and not just scalars.

- Edges will often “skip” layers;
  “layer” is therefore ambiguous.

- Diagram conventions differ;
  e.g., tensorflow graphs include nodes for parameters.
Current-day networks: many layers...

Taken from ResNet paper. 2015.

Taken from Nguyen et al, 2017.
5. pytorch quickstart
Defining networks in pytorch

```python
net1 = torch.nn.Sequential(
    torch.nn.Linear(2, 3, bias = True),
    torch.nn.Linear(3, 4, bias = True),
    torch.nn.Linear(4, 2, bias = True),
)

net2 = torch.nn.Sequential(
    torch.nn.Linear(2, 3, bias = True),
    torch.nn.ReLU(),
    torch.nn.Linear(3, 4, bias = True),
    torch.nn.ReLU(),
    torch.nn.Linear(4, 2, bias = True),
)

for net in (net1, net2):
    print(net(torch.randn(2))) # works
    print(net(torch.randn(1, 2))) # also works
    print(net(torch.randn(10, 2))) # also works
    try:
        print(net(torch.randn(2, 1))) # fails!
    except Exception as e:
        print(e)
```
def fit1(net, X, y, n_epoch = 1000, stepsize = 0.01):
    for epoch in range(n_epoch):
        loss = torch.nn.CrossEntropyLoss()(net(X), y)
        loss.backward()
        with torch.no_grad():
            for P in net.parameters():
                P -= stepsize * P.grad
                P.grad.zero()
        # can alternatively do net.zero_grad()

def fit2(net, X, y, n_epoch = 1000, stepsize = 0.01):
    sgd = torch.optim.SGD(net.parameters(), lr = stepsize)
    for epoch in range(n_epoch):
        loss = torch.nn.CrossEntropyLoss()(net(X), y)
        loss.backward()
        sgd.step()
        sgd.zero_grad()

for net in (net1, net2):
    for fit in (fit1, fit2):
        fit(net, torch.randn(100, 2), torch.randint(2, (100,), dtype = torch.long))
6. Summary (of part 1)
Summary (of part 1)

- Basic deep networks via iterated logistic regression.
- Deep network terminology: parameters, activations, layers, nodes.
- Standard choices: biases, ReLU nonlinearity, cross-entropy loss.
- Basic optimization: magic gradient descent black boxes.
- Basic pytorch code.