pytorch **meta-algorithm.**

1. Clean/augment data (lecture 10?).
2. Pick model/architecture (anything from lectures 2-13).
3. Pick a loss function measuring model fit to data.
4. Run a gradient descent variant to fit model to data.
5. Tweak 1-4 until *training error* is small.
6. Tweak 1-5, *possibly reducing model complexity*, until *testing error* is small.

▶ **Deep networks** have changed how much of machine learning is done, and vastly extended the use of machine learning.

▶ We’ll re-visit the meta-algorithm later to see how things fit together.

▶ **Further reading:** UIUC faculty Svetlana Lazebnik has an entire course with excellent slides on deep learning.
Plan for today

1. Basic deep network definition.
2. Convolutional layers.
3. Other standard layers.
4. Gradient descent and “back-propagation”.
5. pytorch code and meta-algorithm.

(Expect these slides to change before the Tuesday lecture.)

See also my pytorch jupyter notebook [pdf].
(Tutorial will replace Tuesday lecture?)
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.

**Is there an issue with this slide...?**
Wait a minute...

Note that

\[ W_L \left( \cdots \left( W_1 x + b_1 \right) \cdots \right) + b_L \]
\[ = (W_L \cdots W_1) x + (b_L + W_L b_{L-1} + \cdots + W_L \cdots W_2 b_1) \]
\[ = w^\top [x]_1, \]

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

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

Oops, this is just a linear predictor.
1. Activations/nonlinearities.
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.
Iterated logistic regression

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

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

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

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

$$\mathbf{x} \mapsto (f_L \circ \cdots \circ f_1)(\mathbf{x}) \quad \text{where} \quad f_i(z) = \sigma_s(\mathbf{W}_i z + \mathbf{b}_i).$$

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.
A self-contained expression is

\[ x \mapsto \sigma_L \left( W_L \sigma_{L-1} \left( \cdots (W_2 \sigma_1(W_1x + b_1) + b_2) \cdots \right) + b_L \right), \]

with equivalent “functional form”

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

Some further details (many more to come!):

- \((W_i)_{i=1}^L\) with \(W_i \in \mathbb{R}^{d_i \times d_{i-1}}\) are the weights, and \((b_i)_{i=1}^L\) are the biases.
- \((\sigma_i)_{i=1}^L\) with \(\sigma_i : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_i}\) are called nonlinearties, 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:

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

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).

  Original heavy use Fukushima (1960s).

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

- **NON-coordinate-wise choices**: we’ll discuss “softmax” and “pooling”.


Basic form:

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

$$((W_i, b_i))_{i=1}^L$$, the weights and biases, are the parameters.
Let’s roll them into $$\mathcal{W} := ((W_i, b_i))_{i=1}^L$$, 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.)
Remark on affine expansion

Note: we are writing
\[ x \mapsto \sigma_L \left( \cdots \left( W_2 \sigma_1 (W_1 x + b_1) + b_2 \right) \cdots \right), \]
rather than
\[ x \mapsto \sigma_L \left( \cdots \left( W_2 \sigma_1 (W_1 [x]) \right) \cdots \right). \]

- 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?
  Vague belief that it helps...
Classical network/graph perspective

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

\[ v_j := \sigma(z_j), \quad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \quad j \in \{1, 2\}. \]
$v_j := \sigma(z_j)$, \hspace{0.5cm} z_j := \sum_{i=1}^{d} W_{i,j} x_i, \hspace{0.5cm} j \in \{1, \ldots, k\}.$
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.
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 $(w, b)$. 

$\sigma$: activation function, typically a logistic or sigmoid function.
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.

Current “computation graph” perspective.

- Edges can pass full tensors, which helps with current parallel hardware.
- Nodes are more general primitives, not only in the form $\sigma(W z + b)$.
- 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.
\[ \text{Many parameters...} \]

Taken from “attention is all you need”. (Transformer paper, Vaswani et al., 2017.)

Taken from GPT-3 paper. (Brown et al., 2020.)
Which architecture?

How do choose an architecture?

▶ How to choose C and kernel in SVM?
▶ 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 have some preferred architectures. Even these have many attributes which may or may not help.
▶ For new problems and new domains, often there is extensive experimentation with architecture.
Which architecture?

How do choose an architecture?

- How to choose $C$ and kernel in SVM?
Which architecture?

How do choose an architecture?

▶ How to choose $C$ and kernel in SVM?

▶ 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 to choose $C$ and kernel in SVM?
- 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 have some preferred architectures. Even these have many attributes which may or may not help.
- For new problems and new domains, often there is extensive experimentation with architecture.
Convolutional layers

[16 / 52]
We’re missing a few:

- Convolutions.
- Pooling.
- Skip connections.
- Batch norm.

Taken from ResNet paper. 2015.
Continuous convolution in mathematics

- Convolutions are typically continuous:
  \[(f \ast g)(x) := \int f(y)g(x - y) \, dy.\]

- Often, \(f\) is 0 or tiny outside some small interval; e.g., if, \(f\) is 0 outside \([-1, +1]\), then
  \[(f \ast g)(x) = \int_{-1}^{+1} f(y)g(x - y) \, dy.\]

Think of this as sliding \(f\), a filter, along \(g\).
We can also consider discrete convolutions:

\[(f * g)(n) = \sum_{i=-\infty}^{\infty} f(i)g(n - i)\]

If both \(f\) and \(g\) are 0 outside some interval, we can write this as matrix multiplication:

\[
\begin{bmatrix}
  f(1) & 0 & \cdots & \\
  f(2) & f(1) & 0 & \cdots \\
  f(3) & f(2) & f(1) & 0 & \cdots \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  f(d) & f(d-1) & f(d-2) & \cdots & \\
  0 & f(d) & f(d-1) & \cdots & \\
  0 & 0 & f(d) & \cdots & \\
  \vdots & \vdots & \vdots & \ddots & \\
\end{bmatrix}
\begin{bmatrix}
  g(1) \\
  g(2) \\
  g(3) \\
  \vdots \\
  g(m)
\end{bmatrix}
\]

(The matrix at left is a “Toeplitz matrix”.)

Note that we have padded with zeros; the two forms are identical if \(g\) starts and ends with \(d\) zeros.
1-D convolution in deep networks

In PyTorch, this is `torch.nn.Conv1d`.

▶ As above, order reversed wrt “discrete convolution”.

▶ Has many arguments; we’ll explain them for 2-d convolution.

▶ Can also play with it via `torch.nn.functional.conv1d`.
1-D convolution in deep networks

In PyTorch, this is `torch.nn.Conv1d`.

- As above, order reversed wrt "discrete convolution".
- Has many arguments; we'll explain them for 2-d convolution.
- Can also play with it via `torch.nn.functional.conv1d`.
1-D convolution in deep networks

In pytorch, this is `torch.nn.Conv1d`.

▶ As above, order reversed wrt "discrete convolution".

▶ Has many arguments; we’ll explain them for 2-d convolution.

▶ Can also play with it via `torch.nn.functional.conv1d`. 
1-D convolution in deep networks

In PyTorch, this is `torch.nn.Conv1d`.

- As above, order reversed wrt "discrete convolution".
- Has many arguments; we'll explain them for 2-D convolution.
- Can also play with it via `torch.nn.functional.conv1d`.
In PyTorch, this is `torch.nn.Conv1d`.

- As above, order reversed wrt “discrete convolution”.
- Has many arguments; we’ll explain them for 2-d convolution.
- Can also play with it via `torch.nn.functional.conv1d`. 
2-D convolution in deep networks (pictures)

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With padding.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
With padding.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With padding.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With padding.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With padding, strides.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
With padding, strides.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With padding, strides.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With padding, strides.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With dilation.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With dilation.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With dilation.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks (pictures)

With dilation.

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
2-D convolution in deep networks

- Invoke with `torch.nn.Conv2d`, `torch.nn.functional.conv2d`.
- Input and filter can have channels; a color image can have size $32 \times 32 \times 3$ for 3 color channels.
- Output can have channels; this means multiple filters.
- Other `torch` arguments: bias, stride, dilation, padding, . . .
- Was motivated by computer vision community (primate V1); useful in Go, NLP, . . .; many consecutive convolution layers leads to hierarchical structure.
- Convolution layers lead to major parameter savings over dense/linear layers.
- Convolution layers are linear! To check this, replace input $x$ with $ax + by$; the operation to make each entry of output is dot product, thus linear.
- Convolution, like ReLU, seems to appear in all major feedforward networks in past decade!
Other standard layers

Covered here:

▶ Softmax.
▶ Max pooling.
▶ Batch normalization.
▶ Skip connections.

Not covered here:

▶ Other normalization layers.
▶ Blocks, not layers: e.g., attention blocks.
▶ . . .
Softmax

Replace vector input $z$ with $z' \propto e^z$, meaning

$$z \mapsto \left( \frac{e^{z_1}}{\sum_j e^{z_j}}, \cdots, \frac{e^{z_k}}{\sum_j e^{z_j}} \right).$$

- Converts input into a probability vector.
- Typically the final layer of a network; useful for interpreting output network output as $\Pr[Y = y | X = x]$.
  Correspondingly, the inputs to this final softmax are sometimes called logits.
- If some coordinate $j$ of $z$ dominates others, then softmax is close to $e_j$. 
Cross-entropy loss with softmax baked in

Given two probability vectors \( p, q \in \Delta_d := \{ \mathbf{v} \in \mathbb{R}^k : \mathbf{v} \geq 0, \sum_i v_i = 1 \} \), define **cross-entropy**

\[
H(p, q) := -\sum_{i=1}^{k} p_i \ln q_i.
\]

In PyTorch, `cross_entropy(yhat, y)` takes logits \( \hat{y} \in \mathbb{R}^k \) and class label \( y \in [k] \), and computes

\[
\ell_{ce}(\hat{y}, y) = H(e_y, \text{softmax}(\hat{y})) = \ln \left( \frac{\sum_i \exp(\hat{y}_i)}{\exp(\hat{y}_y)} \right)
\]

\[
= -\hat{y}_y + \ln \left( \sum_i \exp(\hat{y}_i) \right).
\]
Max pooling

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
Max pooling

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
Max pooling

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)
Max pooling

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

- Often used together with convolution layers; **shrinks/downsamples** the input.
- Another variant is **average pooling**.
- Implementation: `torch.nn.MaxPool2d`.
Standardize node outputs:

\[ x \mapsto \frac{x - \mathbb{E}(x)}{\text{stddev}(x)} \cdot \gamma + \beta, \]

where \((\gamma, \beta)\) are trainable parameters.

- Standard setups either estimate the mean and variance with each minibatch (including on testing data!) or maintain a running average across minibatches and freeze statistics at test time.

- \((\gamma, \beta)\) defeat the purpose; I’m not sure what values they take on in practice? I used to think they are small?

- No one currently seems to understand batch normalization; (google “deep learning alchemy” for fun;) annecdotally, it sometimes speeds up training and improves generalization.

- It is currently standard in vision architectures.

- In pytorch it’s implemented as a layer; e.g., you can put \texttt{torch.nn.BatchNorm2d} inside \texttt{torch.nn.Sequential}. \textbf{Note:} you must switch the network into \texttt{.train()} and \texttt{.eval()} modes.
Can model resnet as a sequence of blocks computing

\[ z \mapsto z + f_i(z), \]

where a typical \( f_i \) is convolution, batchnorm, relu convolution, relu.

The idea is that \( f_i \) can be initialized small, and each layer is roughly identity;
i.e., the extra layers aren’t making things worse.
Training now tries to improve upon this baseline.

These \( f_i \)'s are residuals.

The identity connections are sometimes called “skip connections”.

There are many variants of the idea (e.g., DenseNet).
Don’t worry about the details too much,
we’ll have a concrete version in hw3.
Deep networks are generally trained with gradient-based methods. We will only give an abstract presentation, since PyTorch lets you not worry about it:

```python
risk.backward()  # compute gradients for all parameters
```

There is little understanding of why this works (results from convex optimization lectures do not apply). Not even “gradient” makes sense (networks in general not differentiable).
Deep networks are generally trained with gradient-based methods.

We will only give an abstract presentation, since PyTorch lets you not worry about it:

```
  risk.backward()  # compute gradients for all parameters
```

There is little understanding of why this works (results from convex opt lectures do not apply).
Not even “gradient” makes sense (networks in general not differentiable).
Warm-up: gradients for linear prediction

Suppose

\[ \hat{R}(w) := \frac{1}{n} \sum_{i=1}^{n} \ell(y_i x_i^T w). \]

▶ Gradient passes through average; focus on one example:

\[ \nabla_w \ell(y x^T w). \]

▶ By the chain rule,

\[ \nabla_w \ell(y x^T w) = \ell'(y x^T w) y x. \]

▶ Abstraction: define \( p(w) := y x^T \), and note

\[ \nabla \ell(y x^T w) = \nabla \ell(p(w)) = \frac{\partial \ell(p(w))}{\partial p(w)} \frac{\partial p(w)}{\partial w}. \]

Let’s iterate this to handle deep networks.
Fix example \((x, y)\), and define

\[
\begin{align*}
p_1 & := \sigma_1(W_1 x), \\
p_{i+1} & := \sigma_{i+1}(W_{i+1} p_i), \\
p_L & := \ell(y \sigma_L(W_L p_{L-1})).
\end{align*}
\]

Then

\[
\nabla W_i \ell(y f(x; w)) = \frac{\partial p_L}{\partial p_i} \frac{\partial p_i}{\partial W_i},
\]

where

\[
\frac{\partial p_L}{\partial p_i} = \frac{\partial p_L}{\partial p_{i+1}} \frac{\partial p_{i+1}}{\partial p_i}.
\]

- **Forward pass:** compute \(p_L\) by computing \(p_1\) and moving up inductively.
- **Backward pass:** compute gradients from \(W_L\) down to \(W_1\) via the inductive chain rule.
- Some people say “backprop is just chain rule”, but in fact it is the computational suggestion to use this two-stage forward/backward recursion.
- Same procedure generalizes to any acyclic computation graph (skip connections, etc.).
Vanishing/exploding gradients

Note the term

$\nabla_{W_i} \ell(y_f(x; \mathbf{w})) = \frac{\partial p_L}{\partial p_i} \frac{\partial p_i}{\partial W_i} = \prod_{j=i}^{L-1} \frac{\partial p_{j+1}}{\partial p_j}$.

With many-layered networks, this computation easily explodes (becomes large) or vanishes (goes to zero).

Many architecture and optimization choices are designed to mitigate this.
“Did you compute your gradients correctly?” — Leon Bottou.
Standard deep networks do not initialize $W_i = 0$.

- **Need to break symmetry**: initialize in some way so that all gradients different.

- **Standard choice for** $W_i \in \mathbb{R}^{d_i \times d_i - 1}$ is $\mathcal{N}(0, 1/d_i - 1)$ per coordinate.

- **Many other tricks**;
  - e.g., batch norm initialization used to force identity residual blocks.
  - Uniform and “truncated Gaussian” are common.
Learning rates

- Lots of magic here. Often $\eta \in \{0.001, 0.01, 0.1\}$ suffices.
- Some use exponentially decaying rates, or rates shaped like a triangle (!). See `torch.optim.lr_scheduler`.
Other optimization details

- Minibatch size seems magic and hardware/problem dependent.
- Terminology: “epoch” is a full pass over training data, but an iteration or step is the update with a single minibatch.
- In addition to SGD, there’s Adam, AdaGrad, AdaDelta, and many others. See torch.optim for many choices.
Data augmentation is a standard part of deep network training:

- Rather than training on just the provided training set, use a bunch of computation to generate new images.
- For example, for vision tasks, it’s common to include random crops and flips, and a few others.
Computation graph diagrams

```
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),
)
```

Remarks.

- Diagram format is not standard.
- As long as anyone can unambiguously reconstruct the network, it's fine.
- Remember that edges can transmit full tensors now!
Other computation graphs...

Taken from ResNet paper, 2015.

AlexNet paper, 2012.
pytorch quickstart

(In an earlier version of the slides, I had a pytorch "quickstart" here. This has been expanded vastly and made into the pytorch tutorial.)
(In an earlier version of the slides, I had a pytorch “quickstart” here. This has been expanded vastly and made into the pytorch tutorial.)
1. Clean/augment data.
   Data augmentation crucial in deep networks; see lecture 10.

2. Pick model/architecture.
   ResNet, DenseNet, …

3. Pick a loss function measuring model fit to data.
   Typically cross-entropy for classification, but many sophisticated choices in modern networks.

4. Run a gradient descent variant to fit model to data.
   Adam and SGD most common, both with minibatches.

5. Tweak 1-4 until training error is small.

6. Tweak 1-5, possibly reducing model complexity, until testing error is small.
   “Model complexity” is not well understood with deep networks; e.g., increasing width can reduce test error.
Summary for today

1. Basic deep network definition.
2. Convolutions and other modern standards.
3. Gradient descent and “back-propagation”.
4. pytorch code and meta-algorithm.
Adversarial examples: on some vision tasks, these networks seem on par with human perception (in terms of training and test error). However, there training points which can be imperceptibly perturbed so that the class label flips! In this way, they are nothing like human perception. Since deep networks are rolling out in many human-facing applications, these examples are scary, and constitute a major area of research.
Adversarial examples: on some vision tasks, these networks seem on par with human perception (in terms of training and test error). However, there are training points which can be imperceptibly perturbed so that the class label flips! In this way, they are nothing like human perception.

Since deep networks are rolling out in many human-facing applications, these examples are scary, and constitute a major area of research.

Feature extraction: we can train a network on some huge data, chop it in the middle, and use these features as input to train a network on some other task, in particular one with much less data.

(The deep learning community sometimes calls this transfer learning; which more generally means transferring information from one prediction task to another.)
Recurrent networks (RNNs). What should we do if our input is some arbitrary length sequence \((x_1, \ldots, x_l)\), e.g., an english sentence? We can have a network which eats this sequence one by one; for \(x_i\), it also consumes a previous state \(s_i\), and outputs \(s_{i+1}\).

Many natural language processing (NLP) tasks now use RNNs.
Recurrent networks (RNNs). What should we do if our input is some arbitrary length sequence \((x_1, \ldots, x_l)\), e.g., an english sentence? We can have a network which eats this sequence one by one; for \(x_i\), it also consumes a previous state \(s_i\), and outputs \(s_{i+1}\). Many natural language processing (NLP) tasks now use RNNs.

Dynamic networks and differentiable programming. In the early code subclassing torch.nn.Module, we could have made the forward function do something more complicated; e.g., the number of layers can be variable.

In pytorch, differentiable programming can concretely mean forward functions that look closer to full Turing Machines.
Recurrent networks (RNNs). What should we do if our input is some arbitrary length sequence \((x_1, \ldots, x_l)\), e.g., an English sentence? We can have a network which eats this sequence one by one; for \(x_i\), it also consumes a previous state \(s_i\), and outputs \(s_{i+1}\). Many natural language processing (NLP) tasks now use RNNs.

Dynamic networks and differentiable programming. In the early code subclassing `torch.nn.Module`, we could have made the forward function do something more complicated; e.g., the number of layers can be variable.

In PyTorch, differentiable programming can concretely mean forward functions that look closer to full Turing Machines.

Architecture search. Since the original work on neural networks, there have been attempts to automatically search for architectures. The bottom line is that it seems such methods waste computation when compared with simple trying 5-10 architectures and training them longer; but maybe it will change.
GPUs can process thousands of simple floating point operations in parallel, and massively speed up many of the computations here (my GPU machine is 100x faster than my laptop when I set things up correctly). In pytorch, you can send torch.nn.Module instances to GPU with .cuda() or .to(), just as with tensors. GPUs are fast when you feed them big tensor operations. (E.g., write ((X @ w - y).norm() *** 2).mean(), not a loop.) Moving things between CPU and GPU is slow.
GPUs can process thousands of simple floating point operations in parallel, and massively speed up many of the computations here (my GPU machine is 100x faster than my laptop when I set things up correctly). In pytorch, you can send torch.nn.Module instances to GPU with .cuda() or .to(), just as with tensors. GPUs are fast when you feed them big tensor operations. (E.g., write ((X @ w - y).norm() *** 2).mean(), not a loop.) Moving things between CPU and GPU is slow.

Dropout is a regularization technique that involves randomly zeroing the outputs of nodes during training. It is less popular than it used to be, but still in use for certain applications (e.g., NLP).
GPUs can process thousands of simple floating point operations in parallel, and massively speed up many of the computations here (my GPU machine is 100x faster than my laptop when I set things up correctly). In PyTorch, you can send `torch.nn.Module` instances to GPU with `.cuda()` or `.to()`, just as with tensors. GPUs are fast when you feed them big tensor operations. (E.g., write `(X @ w - y).norm() ** 2).mean()`, not a loop.) Moving things between CPU and GPU is slow.

Dropout is a regularization technique that involves randomly zeroing the outputs of nodes during training. It is less popular than it used to be, but still in use for certain applications (e.g., NLP).

It is typically stated that deep networks are data hungry. I’m not sure if that’s a necessity, or merely a consequence of our current training practices.
History. Deep networks date back to the 1940s; the original “training algorithms” consisted of a human manually setting weights. They have come and gone multiple times. This phase is the first time they were reliably trainable with so many layers. I’m not sure why, but the reasons include: access to more data, GPUs (ResNet training is very slow), ReLU, random initialization, “social programming” and a generally healthy software ecosystem, . . .
NTK? Neocognitron?
concrete pytorch computation graph
Supplemental reading

The new edition of the Murphy book (available online) has three chapters covering some of these deep network topics, and a few not covered here.