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.
Part 2...
7. Convolutional networks
Continuous convolution in mathematics

- Convolutions are typically continuous:

\[(f * g)(x) := \int f(y)g(x - y) \, dy.\]

- Often, \(f\) is 0 or tiny outside some small interval; e.g., if, \(f\) is 0 outside \([-1, +1]\), then

\[(f * g)(x) = \int_{-1}^{+1} f(y)g(x - y) \, dy.\]

Think of this as sliding \(f\), a filter, along \(g\).
Discrete convolutions in mathematics

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:

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

(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.)
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.

(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 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!
8. Other gates
Softmax

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

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

- Converts input into a probability vector; useful for interpreting output network output as $\Pr[Y = y | X = x]$.
- We have baked it into our cross-entropy definition; last lectures networks with cross-entropy training had implicit softmax.
- If some coordinate $j$ of $z$ dominates others, then softmax is close to $e_j$. 
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

(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>

<table>
<thead>
<tr>
<th>3.0</th>
<th>3.0</th>
<th>3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>3.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

(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:

$$x \rightarrow x - \frac{\mathbb{E}(x)}{\text{stddev}(x)} \cdot \gamma + \beta,$$

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

- $(\gamma, \beta)$ defeat the purpose, but it seems they stay small.
- No one currently seems to understand batch normalization; (google “deep learning alchemy” for fun;) annecdotally, it 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 `torch.nn.BatchNorm2d` inside `torch.nn.Sequential`. **Note:** you must switch the network into `.train()` and `.eval()` modes.
9. Standard architectures
Basic networks (from last lecture)

Input

```
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 someone can unambiguously reconstruct the network, it’s fine.
- Remember that edges can transmit full tensors now!
AlexNet

Oof...
(A variant of) AlexNet

class AlexNet(torch.nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2),
            torch.nn.Conv2d(64, 192, kernel_size=3, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2),
            torch.nn.Conv2d(192, 384, kernel_size=3, padding=1),
            torch.nn.ReLU(),
            torch.nn.Conv2d(384, 256, kernel_size=3, padding=1),
            torch.nn.ReLU(),
            torch.nn.Conv2d(256, 256, kernel_size=3, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2),
        )
        self.classifier = torch.nn.Sequential(
            # torch.nn.Dropout(),
            torch.nn.Linear(256 * 2 * 2, 4096),
            torch.nn.ReLU(),
            # torch.nn.Dropout(),
            torch.nn.Linear(4096, 4096),
            torch.nn.ReLU(),
            torch.nn.Linear(4096, 10),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 2 * 2)
        x = self.classifier(x)
        return x
ResNet

Taken from ResNet paper. 2015.

Taken from Nguyen et al, 2017.
ResNet

- 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.
10. Other topics
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.
GPU 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).
GPU 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 data 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, ...
11. Summary of part 2
Summary of part 2

- Convolutional networks (CNNs).
- Softmax, max-pooling, batch norm.
- General scheme of modern architectures (many layers, many convolutions, skip connections).