Deep Learning 2

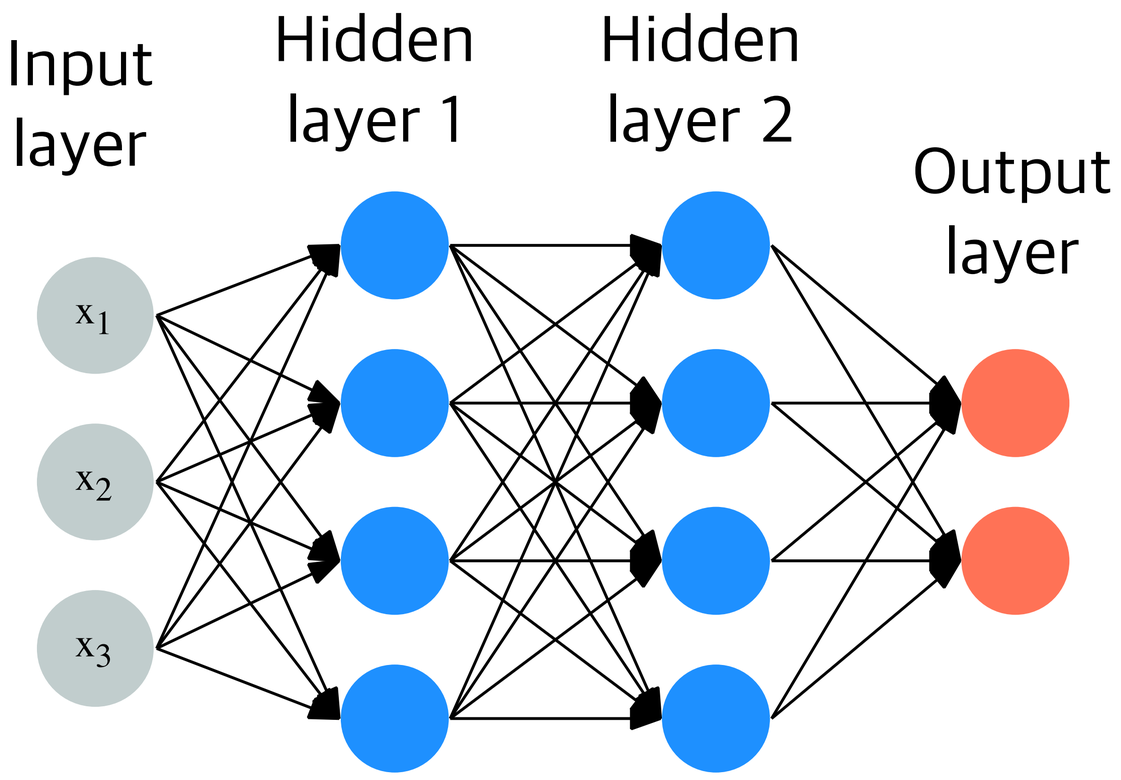
Takashi Ishida
ishi@k.u-tokyo.ac.jp
http://www.ms.k.u-tokyo.ac.jp

Schedule

- 04/11 Introduction
- 04/18 Regression 1: least squares regression
- 04/25 Regression 2: sparse learning & robust learning
- 05/09 Classification 1: least squares classification
- 05/16 Classification 2: support vector classification & probabilistic classification
- 05/23 Deep learning 1: MLPs, backprop, optimizers, regularizers
- 06/06 Deep learning 2: initialization, normalization, NNs for images
- 06/13 Deep learning 3: NNs for sequential data
- 06/20 Semi-supervised learning
- 06/27 Transfer learning
- 07/04 Dimensionality reduction: unsupervised algorithms
- 07/11 Dimensionality reduction: supervised algorithms
- 07/18 Advanced topics

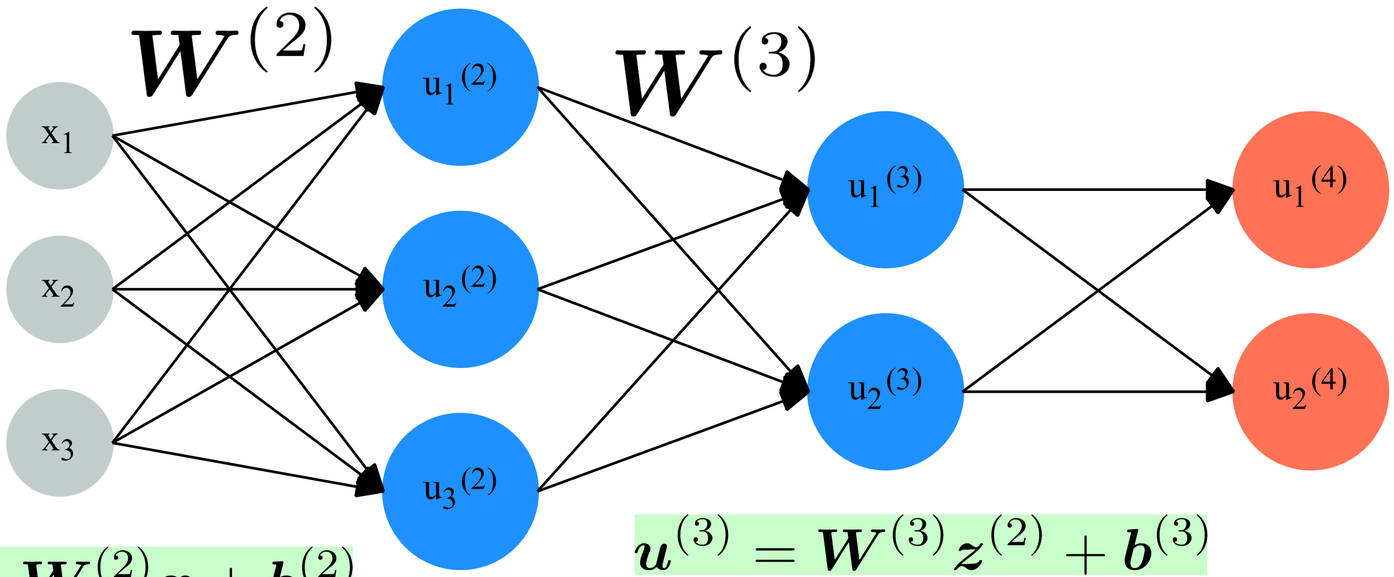
Nonlinear models and neural networks

- Nonlinear model: a model that is nonlinear w.r.t. parameters
- Neural network model: a model that is hierarchical
- Deep neural network model: a model with many layers. Tend to have better generalization and we are seeing more applications.



Multilayer neural net

- Generalize by writing the layer number on upper-right.



$$\boldsymbol{u}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{x} + \boldsymbol{b}^{(2)}$$

$$\boldsymbol{z}^{(2)} = \boldsymbol{a}(\boldsymbol{u}^{(2)})$$

$$\boldsymbol{u}^{(3)} = \boldsymbol{W}^{(3)}\boldsymbol{z}^{(2)} + \boldsymbol{b}^{(3)}$$

$$\boldsymbol{z}^{(3)} = \boldsymbol{a}(\boldsymbol{u}^{(3)})$$

$$\boldsymbol{u}^{(l+1)} = \boldsymbol{W}^{(l+1)}\boldsymbol{z}^{(l)} + \boldsymbol{b}^{(l+1)}$$

$$\boldsymbol{z}^{(l+1)} = \boldsymbol{a}(\boldsymbol{u}^{(l+1)})$$

Loss function for classification

- Training data: $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_{n=1}^N$

  $\boldsymbol{x} \in \mathbb{R}^d$ $\boldsymbol{y} \in \{0,1\}^K$: element becomes 1 only when that element specifies the correct class

  K: # of classes

- Use softmax function after the output layer.
  - The k-th unit in the output layer:

    Satisfies: $\sum_{k=1}^K f_k = 1$

$$f_k \equiv z_k^{(L)} = \frac{\exp(u_k^{(L)})}{\sum_{j=1}^K \exp(u_j^{(L)})}$$

Note: this is what we covered in the previous lecture!

  - Loss function: cross entropy

$$J(\boldsymbol{w}) = -\sum_{n=1}^N \sum_{k=1}^K y_{nk} \log f_k(\boldsymbol{x}_n; \boldsymbol{w})$$

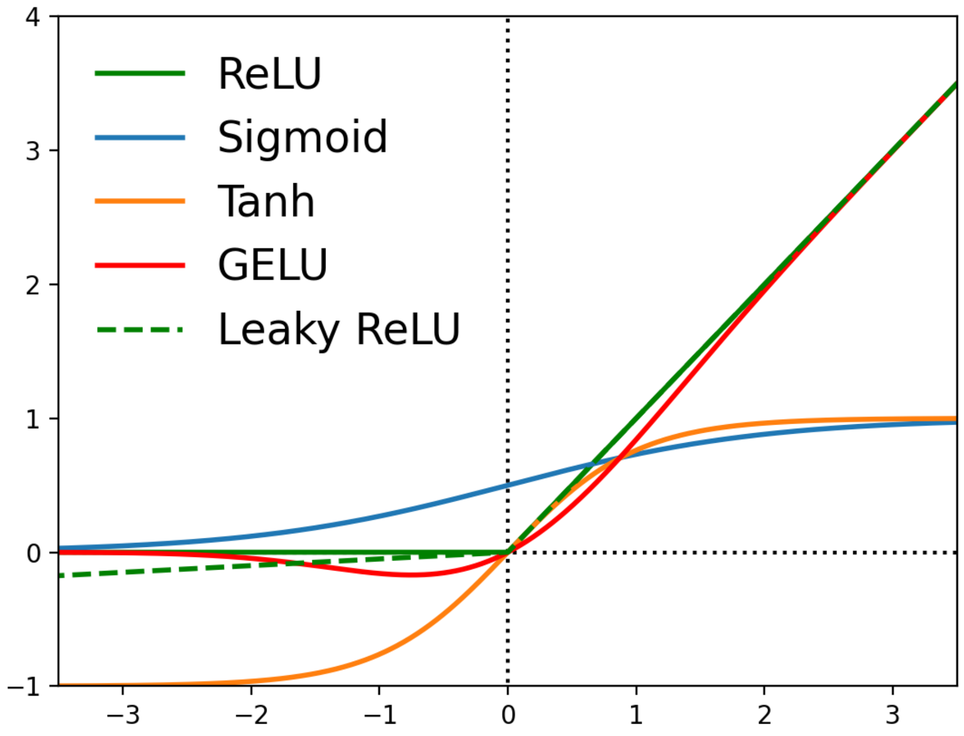
Activation function

$$\boldsymbol{z}^{(l+1)} = \boldsymbol{a}(\boldsymbol{u}^{(l+1)})$$

- Various activation functions have been proposed.
- Many of them are monotonically increasing, differentiable, and nonlinear.
- It's also possible to use different activation functions for each layer.

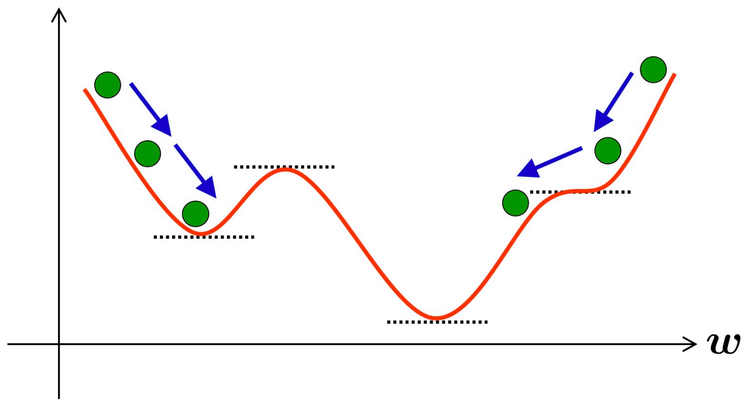
Name	July 2021	May 2023
ReLU	5638	8186
Sigmoid Activation	3896	5527
Tanh Activation	3683	5068
GELUs	2665	5764
Leaky ReLU	510	973

Number of papers from Papers with Code:
https://paperswithcode.com/methods/category/activation-functions



Gradient descent method

- Ideally, we want: $\boldsymbol{w} = \mathrm{argmin}_{\boldsymbol{w}} J(\boldsymbol{w})$.
- However, $J(\boldsymbol{w})$ is generally not convex.
  - Usually impossible to directly obtain the global minimum solution.
- Obtain a local minimum point using the gradient descent method.
  - Start from initialized parameters.
  - Gradient is: $\nabla J \equiv \dfrac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = \left[\dfrac{\partial J}{\partial w_1} \quad \cdots \quad \dfrac{\partial J}{\partial w_M} \right]^\top$
  - Repeat update: $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \epsilon \nabla J(\boldsymbol{w})$
    - ϵ: step size (learning rate)
  - Repeat until stopping condition is met.



Deeply nested problem

$$\frac{\partial J_n}{\partial w_{ji}^{(l)}} = (\boldsymbol{f}(\boldsymbol{x}_n) - \boldsymbol{y}_n)^\top \frac{\partial \boldsymbol{f}}{\partial w_{ji}^{(l)}}$$

- $\boldsymbol{W}^{(l)}$ appears in:

$$\begin{aligned}
&\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{a}\left(\boldsymbol{u}^{(L)}\right) = \boldsymbol{a}\left(\boldsymbol{W}^{(L)}\boldsymbol{z}^{(L-1)} + \boldsymbol{b}^{(L)}\right) \\
&= \boldsymbol{a}\left(\boldsymbol{W}^{(L)}\boldsymbol{a}\left(\boldsymbol{W}^{(L-1)}\boldsymbol{z}^{(L-2)} + \boldsymbol{b}^{(L-1)}\right) + \boldsymbol{b}^{L}\right) \\
&= \boldsymbol{a}\left(\boldsymbol{W}^{(L)}\boldsymbol{a}\left(\boldsymbol{W}^{(L-1)}\boldsymbol{a}\left(\cdots \boldsymbol{a}\left(\boldsymbol{W}^{(l)}\boldsymbol{z}^{(l-1)} + \boldsymbol{b}^{(l)}\right)\cdots\right)\right) + \boldsymbol{b}^{(L)}\right)
\end{aligned}$$

  - We need to apply the chain rule many times!
  - Implementation becomes more difficult and complicated.
- Use backpropagation

Implementation part 1 (see ITC-LMS for full code)


```
import torch; import torch.nn.functional as F
from torch.utils.data import DataLoader; from torchvision import datasets, transforms

torch.manual_seed(0)
lr = 0.005; hidden_dim = 500; batch_size = 8; epochs = 5

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=1000, shuffle=False)

class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        self.W1= torch.randn(input_size, hidden_size) * 0.1
        self.b1= torch.randn(hidden_size) * 0.1
        self.W2= torch.randn(hidden_size, output_size) * 0.1
        self.b2= torch.randn(output_size) * 0.1

    def forward(self, x):
        self.x = x
        self.z1 = x @ self.W1 + self.b1
        self.a1 = F.relu(self.z1)
        self.z2 = self.a1 @ self.W2 + self.b2
        return self.z2

model = TwoLayerNet(784, hidden_dim, 10)
```

one-hidden layer neural net w/ ReLU

(we will later discuss about neural network initialization in detail)

forward pass

Implementation part 2 (see ITC-LMS for full code)


```python
def train(epoch):
    for batch_idx, (data, target) in enumerate(train_loader):
        data = data.view(-1, 784)  # flatten the input

        # === FORWARD PASS ===
        output = model.forward(data)
        log_softmax = F.log_softmax(output, dim=1)
        loss = - torch.mean(log_softmax[range(len(target)), target]) # =NLLLoss
        # === BACKWARD PASS ===
        # gradient of the loss w.r.t. output of model
        grad_z2 = F.softmax(output, dim=1)
        grad_z2[range(len(target)), target] -= 1
        grad_z2 /= len(target) # recall that loss is average over batch
        # gradient of the loss w.r.t. the output after ReLU
        grad_a1 = grad_z2 @ model.W2.T
        # gradient of the loss w.r.t. the output before ReLU
        grad_z1 = grad_a1.clone()
        grad_z1[model.z1 < 0] = 0
        # gradient of the loss w.r.t. the model parameters
        model.W2.grad = model.a1.T @ grad_z2
        model.b2.grad = grad_z2.sum(axis=0)
        model.W1.grad = model.x.T @ grad_z1
        model.b1.grad = grad_z1.sum(axis=0)
        # === PARAM UPDATES ===
        model.W2 = model.W2 - lr * model.W2.grad
        model.b2 = model.b2 - lr * model.b2.grad
        model.W1 = model.W1 - lr * model.W1.grad
        model.b1 = model.b1 - lr * model.b1.grad
```

forward pass

backward pass

prepare gradient of parameters

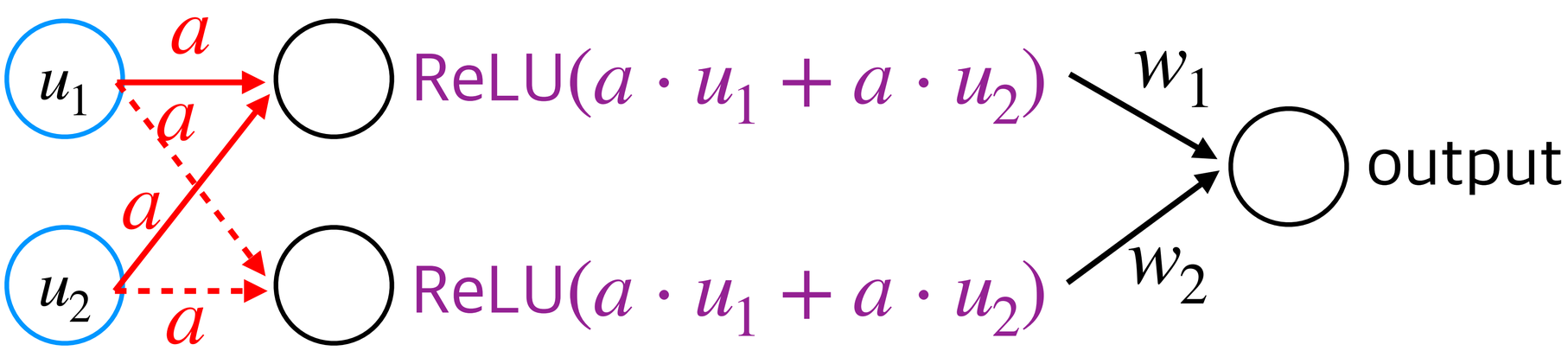
vanilla SGD

See a PyTorch-ified version and observe the differences: https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

Contents

How should we initialize neural nets?

- We need to be careful about how we initialize the parameters.
- Example: if weights before hidden layer is the same, e.g., all zero, then the hidden nodes will all have the same value.



- The local gradient w.r.t. the red params will all be the same.
- The gradient signal coming back from the output to the hidden unit will be different because $w_1 \neq w_2$. The two dense arrows (or dotted arrows) will have the same update.

Initialize W1's weights to 0

- In last week's code, if we initialize W1 (input_dim × hidden_dim) to 0 and train for 1 epoch:

```
model.W1
tensor([[-0.0022, -0.0009, -0.0014,  ..., -0.0005],
        [-0.0022, -0.0009, -0.0014,  ..., -0.0005],
        [-0.0022, -0.0009, -0.0014,  ..., -0.0005],
        ...,
        [-0.0022, -0.0009, -0.0014,  ..., -0.0005],
        [-0.0022, -0.0009, -0.0014,  ..., -0.0005],
        [-0.0022, -0.0009, -0.0014,  ..., -0.0005]])
```

```
model.W2
tensor([[-0.1655, -0.2811, -0.0117, -0.2020,  0.2901,  0.0185,  0.3528, -0.1439,  0.3307,  0.1754],
        [ 0.4580, -0.3907, -0.0706,  0.0204, -0.0394,  0.2145, -0.1748, -0.0833,  0.2144,  0.0095],
        [ 0.1333,  0.1968, -0.0949, -0.0179, -0.3295,  0.2614, -0.0619, -0.3549, -0.1177, -0.1976],
        [-0.0454,  0.0048,  0.1085, -0.2594,  0.2418,  0.0422,  0.1974,  0.0916, -0.3027, -0.1114],
        [ 0.2399, -0.2905,  0.3678,  0.1251, -0.1953, -0.0255,  0.1946, -0.0235, -0.0873, -0.1273],
        [-0.0538,  0.3236, -0.0417,  0.2776, -0.0898, -0.1139, -0.0275, -0.1133,  0.3653, -0.1949],
        [-0.0044, -0.0687,  0.2155,  0.1019,  0.3654, -0.0111,  0.0424, -0.2344,  0.1459, -0.0824],
        [-0.0985, -0.1674, -0.0670,  0.2151, -0.1783,  0.1443, -0.1338,  0.2785, -0.1409,  0.2713],
        [-0.0801,  0.1066,  0.3436,  0.0044, -0.0475,  0.0473,  0.5059,  0.0078, -0.1274, -0.1050],
        [-0.3367,  0.3481,  0.1072,  0.4021, -0.0041, -0.0460, -0.2105,  0.0365,  0.1188,  0.1687],
        [-0.1675,  0.2172,  0.2605, -0.1983, -0.1114, -0.1353, -0.1217,  0.2612,  0.1069,  0.1397],
        [ 0.1468, -0.0645,  0.1487, -0.0769,  0.0234,  0.0568, -0.0656, -0.0798, -0.0500,  0.1314],
        [-0.1590, -0.2445, -0.2201,  0.0149,  0.0634,  0.3418, -0.0789,  0.2298,  0.0250, -0.1623],
        [-0.0396, -0.0355, -0.0365, -0.0245,  0.2920, -0.0010, -0.0528, -0.1364, -0.0684,  0.4783],
        [ 0.0892,  0.1813, -0.2559, -0.2971,  0.2006, -0.3014,  0.1895,  0.3720, -0.1066,  0.1163]])
```

- 👀 Observe W1:
  - All hidden units connected to the same input unit has the same value! (see each column)
  - Wasting a lot of parameters in W1: 784 x 500 = 392,000 —> 500
  - We need to break the symmetry.
- 👀 Observe W2:
  - Looks fine. This is because we initialized randomly for W2.

Initialize with Gaussian distribution?

- If we initialize with $\mathcal{N}(0,1)$, the loss after initialization is 815 (MNIST; 10 classes).
- With more layers, this can become more severe.
- Large loss implies high probabilities for wrong predictions. In the 1st epoch, loss goes from 815 to 23. We are squashing those high probabilities in the early stage of learning.
- However, if we multiply the params by 0.1: loss initialized to 8.46
- Looks much better, but are there still room for improvement? What loss value should we aim for before training (right after initialization)?

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        self.W1= torch.randn(input_size, hidden_size) * 1 #0.1
        self.b1= torch.randn(hidden_size) * 1 #0.1
        self.W2= torch.randn(hidden_size, output_size) * 1 #0.1
        self.b2= torch.randn(output_size) * 1 #0.1

    def forward(self, x):
        self.x = x
        self.z1 = x @ self.W1 + self.b1
        self.a1 = F.relu(self.z1)
        self.z2 = self.a1 @ self.W2 + self.b2
        return self.z2
```

***1.0**

```
Train Epoch: 1 [0/60000 (0%)]   Loss: 815.423401
Train Epoch: 1 [12800/60000 (21%)]    Loss: 35.187027
Train Epoch: 1 [25600/60000 (43%)]    Loss: 27.583292
Train Epoch: 1 [38400/60000 (64%)]    Loss: 46.137630
Train Epoch: 1 [51200/60000 (85%)]    Loss: 38.797104
```

***0.1**

```
Train Epoch: 1 [0/60000 (0%)]         Loss: 8.459664
Train Epoch: 1 [12800/60000 (21%)]  Loss: 0.498634
Train Epoch: 1 [25600/60000 (43%)]  Loss: 0.410814
Train Epoch: 1 [38400/60000 (64%)]  Loss: 0.711170
Train Epoch: 1 [51200/60000 (85%)]  Loss: 0.576229
```

Loss to expect before training

- Natural to expect roughly uniform probabilities over the class predictions before training:

$$1/K \text{ for } K \text{ classes}$$

- Since we are using softmax cross-entropy loss function, this means we expect our loss to be:

$$-\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\log f_k(\boldsymbol{x}_n) = -\frac{1}{N}\sum_{n=1}^{N}\log 1/K = -\log(1/K)$$

  - Examples: this will become 2.3 for $K = 10$, 4.6 for $K = 100$, and 6.9 for $K = 1000$

- In order to expect $1/K$ for our class probabilities, we expect our logits to be $[a, a, \ldots, a]$ because softmax$([a, a, \ldots a]) = [1/K, \ldots, 1/K]$.

- Instead of having a positive or negative bias, we now consider $a = 0$ as our target.

Smaller scale or larger scale?

- In the same model, we multiply by 0.0001 so that our logits become close to a zero vector.

close to $-\log(1/K)$!

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        self.W1= torch.randn(input_size, hidden_size) * 0.0001
        self.b1= torch.randn(hidden_size) * 0.0001
        self.W2= torch.randn(hidden_size, output_size) * 0.0001
        self.b2= torch.randn(output_size) * 0.0001

    def forward(self, x):
        ...
```

```
Train Epoch: 1 [0/60000 (0%)]    Loss: 2.302597
Train Epoch: 1 [12800/60000 (21%)]    Loss: 2.302358
Train Epoch: 1 [25600/60000 (43%)]    Loss: 2.298049
Train Epoch: 1 [38400/60000 (64%)]    Loss: 2.276546
Train Epoch: 1 [51200/60000 (85%)]    Loss: 1.922438
train set: Average loss: 1.5085, Accuracy: 32759/60000 (55%)
test set: Average loss: 1.4939, Accuracy: 5526/10000 (55%)
```

- However, learning is super slow. This is because the units in hidden layer all have very similar values. Although symmetry breaking occurs, gradients are similar to each other in the early stage of learning.
- We need to look for a slightly bigger scale that still gives a moderate initial loss.
- 🔍 Manually looking for this will get out of control, especially with many layers!

Investigating mean/std in our network: part 1


```
import torch
import torch.nn.functional as F

input_size = 100; hidden_size = 100

x = torch.randn(input_size)
W1 = torch.randn(input_size, hidden_size)

z1 = x @ W1 # (100) @ (100, 100) --> (100)
a1 = F.relu(z1)

print('{:.2f}, {:.2f}'.format(x.mean().item(), x.std().item())) # 0.04, 1.03
print('{:.2f}, {:.2f}'.format(W1.mean().item(), W1.std().item())) # -0.00, 0.99
print('{:.2f}, {:.2f}'.format(z1.mean().item(), z1.std().item())) # -0.17, 10.68
print('{:.2f}, {:.2f}'.format(a1.mean().item(), a1.std().item())) # 3.96, 6.20
```

std is large!

- Math note: If $x_i \sim \mathcal{N}(0,1)$, $y_i \sim \mathcal{N}(0,1)$, $i \in [n]$, consider $z_n = \sum_{i=1}^{n} x_i y_i$.
  - $\mathbb{E}[z_n] = \mathbb{E}[\sum_{i=1}^{n} x_i y_i] = \sum_{i=1}^{n} \mathbb{E}[x_i]\mathbb{E}[y_i] = 0$
  - $\mathbb{V}[z_n] = \mathbb{E}[(\sum x_i y_i - 0)^2] = \sum \mathbb{E}[x_i^2] \cdot \mathbb{E}[y_i^2] = \sum_{i=1}^{n} 1 \cdot 1 = n$

Investigating mean/std in our network: part 2


```
import torch; import torch.nn.functional as F

input_size = 100; hidden_size = 100
x = torch.randn(input_size)
W1 = torch.randn(input_size, hidden_size) / (input_size ** 0.5)
z1 = x @ W1
a1 = F.relu(z1)

print('{:.2f}, {:.2f}'.format(x.mean().item(), x.std().item())) # 0.04, 1.03
print('{:.2f}, {:.2f}'.format(W1.mean().item(), W1.std().item())) # -0.00, 0.10
print('{:.2f}, {:.2f}'.format(z1.mean().item(), z1.std().item())) # -0.02, 1.07
print('{:.2f}, {:.2f}'.format(a1.mean().item(), a1.std().item())) # 0.40, 0.62
```

adjusted (much smaller)

std is 1

shrinking?

positive bias (due to ReLU!)

- Xavier's initialization (2010; paper): using a normal distribution with std $1/\sqrt{n}$ where n is the number of units in the previous layer.
- Math note: when $x \sim \mathcal{N}(0,1)$ and we transform by $y = ax$: $y \sim \mathcal{N}(0,a^2)$

Investigating mean/std in our network: part 3


```
import torch
import torch.nn.functional as F

input_size = 100; hidden_size = 100; output_size = 10

x = torch.randn(input_size)
W1 = torch.randn(input_size, hidden_size) / ((input_size/2) ** 0.5)

z1 = x @ W1
a1 = F.relu(z1)

print('{:.2f}, {:.2f}'.format(x.mean().item(), x.std().item())) # -0.08, 1.10
print('{:.2f}, {:.2f}'.format(W1.mean().item(), W1.std().item())) # -0.00, 0.14
print('{:.2f}, {:.2f}'.format(z1.mean().item(), z1.std().item())) # -0.04, 1.60
print('{:.2f}, {:.2f}'.format(a1.mean().item(), a1.std().item())) # 0.65, 0.99
```

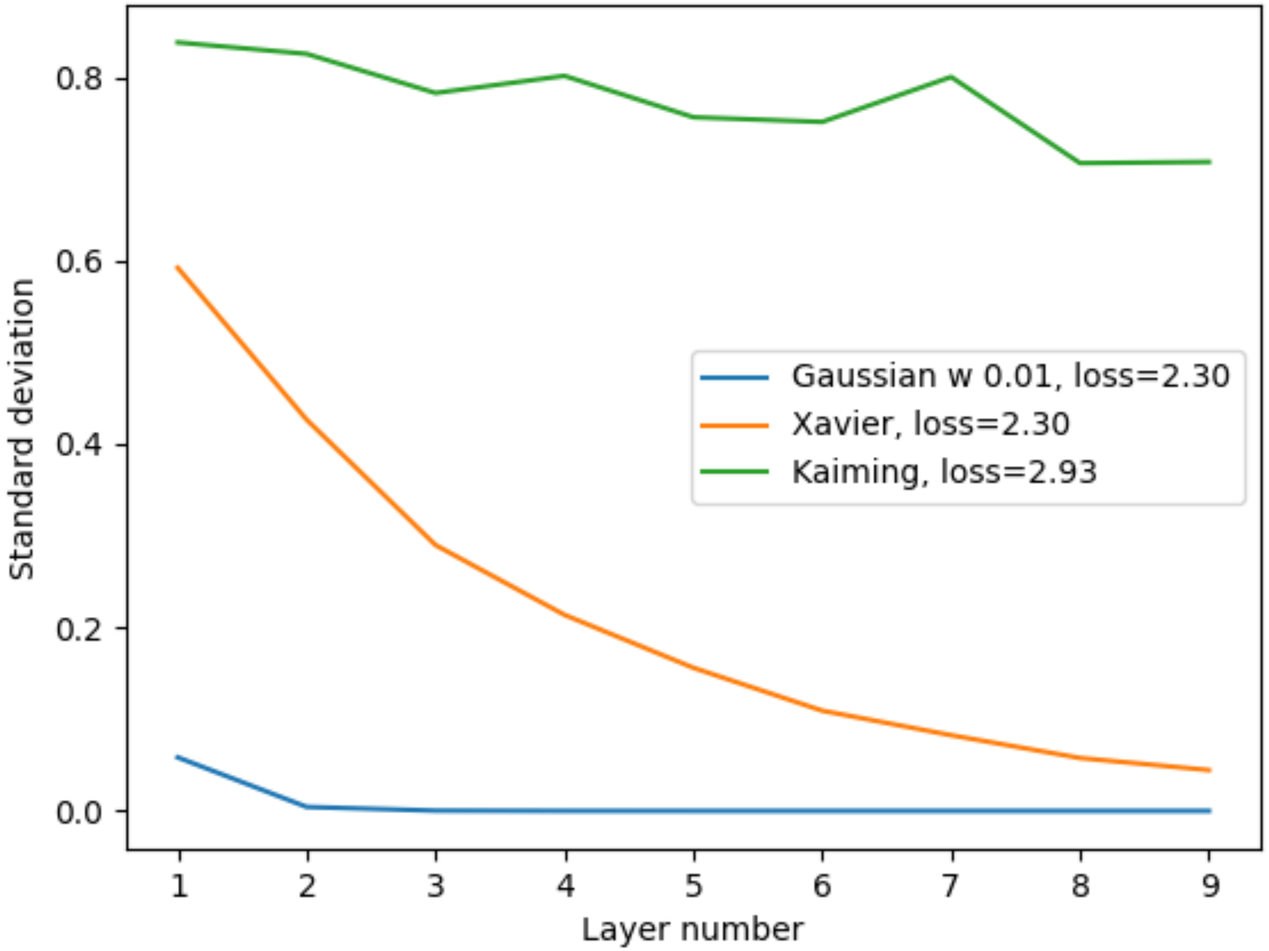
std > 1

std is preserved

- Kaiming's initialization (2015; paper): using a normal distribution with std $\sqrt{2/n}$ where n is the number of units in the previous layer.

Comparing the 3 methods

- Setup
  - We consider 10 layers.
  - For each layer, we plot the mean and std of the values after the ReLU activation.
- Results
  - The standard deviations of "Normal Gaussian * 0.01" and "Xavier" decrease rapidly. The initial loss is good (2.30) but this is just because all weights are small and similar.
  - Although the initial loss is a bit higher, "Kaiming" maintains a moderate level of std even after 10 layers.
- Key takeaway
  - We want to have a good balance of the variance and the initial loss even with many layers.



Contents

Why not normalize each layer directly?

- Instead of being careful about initializing weights, why not just normalize each layer directly so that they have unit mean and variance?
- Standardization is a differentiable operation. We can safely add it to our neural net (we will see this in detail soon.)
- **Idea: normalize in a batch-wise fashion for each units!**
- This was one of the main ideas proposed in "Batch normalization: accelerating deep network training by reducing internal covariate shift" by Ioffe and Szegedy in ICML 2015.

Toy example of normalization in the batch dimension


```
import torch; import torch.nn.functional as F
input_size = 100; hidden_size = 100; batch_size = 32

x = torch.randn(batch_size, input_size)
W1 = torch.randn(input_size, hidden_size) # ignore biases for this example
W2 = torch.randn(hidden_size, hidden_size)
W3 = torch.randn(hidden_size, hidden_size)

z1 = x @ W1
mean1 = z1.mean(0, keepdim=True)
var1 = z1.var(0, keepdim=True)
z1 = (z1 - mean1) / torch.sqrt(var1 + 1e-7); a1 = F.relu(z1)

z2 = a1 @ W2
mean2 = z2.mean(0, keepdim=True)
var2 = z2.var(0, keepdim=True)
z2 = (z2 - mean2) / torch.sqrt(var2 + 1e-7); a2 = F.relu(z2)

z3 = a2 @ W3
mean3 = z3.mean(0, keepdim=True)
var3 = z3.var(0, keepdim=True)
z3 = (z3 - mean3) / torch.sqrt(var3 + 1e-7); a3 = F.relu(z3)

print('{:.2f}, {:.2f}'.format(z1.mean().item(), z1.std().item())) # 0.00, 0.98
print('{:.2f}, {:.2f}'.format(z2.mean().item(), z2.std().item())) # -0.00, 0.98
print('{:.2f}, {:.2f}'.format(z3.mean().item(), z3.std().item())) # -0.00, 0.98
```

We need one more idea…

- In general, normalizing the network will not give good results! Why not?
  - Normalization is helpful from the perspective of initialization.
  - However, it will force the weights to be unit mean and variance throughout training, which may potentially be harmful.
  - We want uniform probabilities only at the beginning!
  - After training, we want to have high probability for the correct class which may require the weights to be far from the normalized values.
- 💡 We allow to re-scale and re-shift.

Batch normalization

- If we have values of a (pre-activation) unit x over a mini-batch $\mathscr{B} = \{x_1, \ldots, x_m\}$, the batch norm operation is:
  - $\mu_{\mathscr{B}} = \frac{1}{m} \sum_{i=1}^{m} x_i$ mini-batch mean
  - $\sigma_{\mathscr{B}}^2 = \frac{1}{m} \sum_{i=1}^{m} \left(x_i - \mu_{\mathscr{B}}\right)^2$ mini-batch variance
  - $\widehat{x}_i = \frac{x_i - \mu_{\mathscr{B}}}{\sqrt{\sigma_{\mathscr{B}}^2 + \epsilon}}$ normalization
  - $u_i = \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}\left(x_i\right)$ scale and shift
- We learn these two parameters along with the weight parameters! (Initialized as $\gamma = 1$ and $\beta = 0$.)

Batch norm: β makes bias term in neural net unnecessary

- Assume our post-activation param vector is:

$$z = g(\boldsymbol{W}\boldsymbol{u} + \boldsymbol{b})$$

$g(\cdot)$ is an activation function

- By adding batch norm pre-activation:

$$z = g\left(\text{BN}(\boldsymbol{W}\boldsymbol{u} + \boldsymbol{b})\right)$$

- Since BN also has a "shifting" operation with $\boldsymbol{\beta}$, we can ignore the bias term $\boldsymbol{b}$:

$$z = g\left(\text{BN}(\boldsymbol{W}\boldsymbol{u})\right)$$

```python
import torch.nn as nn

class mlp_model(nn.Module):
    def __init__(self, input_dim, output_dim, middle_dim=500):
        super(mlp_model, self).__init__()
        self.fc1 = nn.Linear(input_dim, middle_dim, bias=False)
        self.bn1 = nn.BatchNorm1d(middle_dim)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(middle_dim, middle_dim, bias=False)
        self.bn2 = nn.BatchNorm1d(middle_dim)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(middle_dim, middle_dim, bias=False)
        self.bn3 = nn.BatchNorm1d(middle_dim)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(middle_dim, middle_dim, bias=False)
        self.bn4 = nn.BatchNorm1d(middle_dim)
        self.relu4 = nn.ReLU()
        self.fc5 = nn.Linear(middle_dim, output_dim)

    def forward(self, x):
        out = x
        out = self.relu1(self.bn1(self.fc1(out)))
        out = self.relu2(self.bn2(self.fc2(out)))
        out = self.relu3(self.bn3(self.fc3(out)))
        out = self.relu4(self.bn4(self.fc4(out)))
        out = self.fc5(out)
        return out
```

An example of an MLP model with BN written in PyTorch

BN is a differentiable transformation

- We can derive $\frac{\partial J}{\partial x_i}$ (this is one of the homework)
  - If this can be derived, we can also derive the gradients in the earlier layers!
- Math exercise:
  - Since we also want to learn shift/scale parameters in the BN operation, let's derive $\frac{\partial J}{\partial \gamma}$ and $\frac{\partial J}{\partial \beta}$. Assume that we already derived $\frac{\partial J}{\partial u_i}$ (u_i is output of BN layer)
- Recall:
  - $\mu_{\mathscr{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$ mini-batch mean; $\sigma_{\mathscr{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x_i - \mu_{\mathscr{B}}\right)^2$ mini-batch variance
  - $\hat{x}_i = \frac{x_i - \mu_{\mathscr{B}}}{\sqrt{\sigma_{\mathscr{B}}^2 + \epsilon}}$ normalization; $u_i = \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}\left(x_i\right)$ scale and shift

Solution

- Since $u_i = \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}\left(x_i\right)$:
  - $$\frac{\partial J}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial J}{\partial u_i} \cdot \frac{\partial u_i}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial J}{\partial u_i} \cdot \hat{x}_i$$
  - $$\frac{\partial J}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial J}{\partial u_i} \cdot \frac{\partial u_i}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial J}{\partial u_i} \cdot 1 = \sum_{i=1}^{m} \frac{\partial J}{\partial u_i}$$

Batch norm: inference stage

- The architecture is designed to work with a mini-batch.
- In inference stage, we may have different prediction for a test sample depending on what other samples are included in the mini-batch.
- Make this deterministic:
  - After training, use all training data to derive the normalization statistics and normalize: $\hat{x} = (x - \mathbb{E}[x])/\sqrt{\mathbb{V}[x] + \epsilon}$
  - We can now predict for a single sample!
- Alternative idea:
  - Instead, we can keep a running average during training.
  - Skip full forward pass <u>after</u> training, enables val. accuracy check <u>during</u> training

train/eval mode in PyTorch

```
import torch; import torch.nn as nn

class BNNet(nn.Module):
    def __init__(self):
        super(BNNet, self).__init__()
        self.fc1 = nn.Linear(20, 10)
        self.bn1 = nn.BatchNorm1d(10)
        self.fc2 = nn.Linear(10, 1)

    def forward(self, x):
        x = self.fc2(torch.relu(self.bn1(self.fc1(x))))
        return x

x = torch.randn(3, 20) # 3 samples
net = BNNet()

net.train()
#net(x[[0],]) # ValueError: Expected more than 1 value per channel when training

print(net(x[[0,1],])[0], net(x[[0,2],])[0]) # -0.6516, -0.2094

net.eval()
print(net(x[[0],])) # no more errors! # -0.1022
print(net(x[[0,1],])[0], net(x[[0,2],])[0]) # -0.1022, -0.1022
```

Layer normalization

- 🚨 Batch norm cannot be used with a small mini-batch (or a single sample).
- ✅ Alternative idea (called layer normalization; Ba et al. 2016):
  - Assume $\boldsymbol{a} = \boldsymbol{W}\boldsymbol{u} + \boldsymbol{b}$ is the vector before activation with J units.
  - We take mean: $\mu = (1/J)\sum_{j=1}^{J} u_j$ and variance $\sigma^2 = (1/J)\sum_{j=1}^{J}(u_j - \mu)^2$.
  - Normalize by: $\hat{u}_j = \gamma_j \dfrac{u_j - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta_j$
  - γ_j, β_j are learnable parameters.
- Comparing with batch norm
  - BN/LN normalize along the sample/layer dimension, respectively.
  - LN behaves the same for training and test time.

Toy example of layer norm

```
import torch; import torch.nn as nn

class LNNet(nn.Module):
    def __init__(self):
        super(LNNet, self).__init__()
        self.fc1 = nn.Linear(5, 10)
        self.ln1 = nn.LayerNorm(10)
        self.fc2 = nn.Linear(10, 1)

    def forward(self, x):
        x = self.fc2(torch.relu(self.ln1(self.fc1(x))))
        return x

x = torch.randn(1, 5)
net = LNNet()

u = net.fc1(x)
gamma = net.ln1.weight # [1., 1., ..., 1.]
beta = net.ln1.bias # [0., 0., ..., 0.]

# comparing manual layer norm and nn.LayerNorm
z_manual = beta + gamma * (u - u.mean(dim=1, keepdim=True)) \
    / torch.sqrt(u.var(dim=1, keepdim=True, unbiased=False) + 1e-05)
z_pytorch = net.ln1(u)
print(torch.allclose(z_manual, z_pytorch)) # true!

# works with a single sample
print(net(x[[0,],]))
```

- Notice how we are no longer switching between train/eval mode.

Contents

Convolution and Filter

- We slide the filter matrix over the input matrix in a grid-like fashion.
- First multiply element-wise between the filter and the overlaying part of the input matrix, then sum to produce a single output.

```
[[1, 1, 2, 4],
 [5, 6, 7, 8],
 [3, 2, 1, 0],
 [1, 2, 3, 4]]
```

Input data

(4,4)

$\circledast$

```
[[0, 1],
 [2, 3]]
```

Filter/kernel

(2,2)

$\longrightarrow$

- Quick exercise: what goes in "??"

```
[[29., 35., 42.],
 [18., 14., 10.],
 [10., 14., ??.]]
```

Result

(3,3)

Convolution and Filter

- We slide the filter matrix over the input matrix in a grid-like fashion.
- First multiply element-wise between the filter and the overlaying part of the input matrix, then sum to produce a single output.

```
[[1, 1, 2, 4],
 [5, 6, 7, 8],
 [3, 2, 1, 0],
 [1, 2, 3, 4]]
```

Input data

(4,4)

$\circledast$

```
[[0, 1],
 [2, 3]]
```

Filter/kernel

(2,2)

$\longrightarrow$

```
[[29., 35., 42.],
 [18., 14., 10.],
 [10., 14., 18.]]
```

Result

(3,3)

Understanding filters

- Weight sharing:
  - Fully-connected layer: 16 —> 9 will require 144 weight parameters.
  - Filter: only has 4 (learnable) parameters.
- Filters move across the input data to identify certain patterns.

```
[[1., 0., 0., 0.],
 [0., 1., 0., 0.],
 [0., 0., 1., 0.],
 [0., 0., 0., 1.]]
```

$\circledast$

```
[[0, 1],
 [1, 0]]
```

$\longrightarrow$

```
[[0., 1., 0.],
 [1., 0., 1.],
 [0., 1., 0.]]
```

```
[[0., 0., 0., 1.],
 [0., 0., 1., 0.],
 [0., 1., 0., 0.],
 [1., 0., 0., 0.]]
```

$\circledast$

```
[[0, 1],
 [1, 0]]
```

$\longrightarrow$

```
[[0., 0., 2.],
 [0., 2., 0.],
 [2., 0., 0.]]
```

Padding

- We often “pad” by adding extra rows/columns filled with zeros to the borders of a matrix.
- It helps to maintain the spatial size of the output volume as the original input.
- In PyTorch, `torch.nn.functional.pad` can be used to pad a tensor.

```
[[1, 1, 2, 4],
 [5, 6, 7, 8],
 [3, 2, 1, 0],
 [1, 2, 3, 4]]
```

→

```
[[0, 0, 0, 0, 0, 0],
 [0, 1, 1, 2, 4, 0],
 [0, 5, 6, 7, 8, 0],
 [0, 3, 2, 1, 0, 0],
 [0, 1, 2, 3, 4, 0],
 [0, 0, 0, 0, 0, 0]]
```

An example of padding of one unit. We are simply adding one row/column of values (usually zeros) around the input matrix.

Why should we use padding?

- It prevents loss of information from the corners and edges of the input, as these areas would otherwise have fewer filter applications compared to the center of the input.
- For example, the 1 in the input matrix is applied 4 times.

```
[[0, 0, 0, 0, 0, 0],
 [0, 1, 1, 2, 4, 0],
 [0, 5, 6, 7, 8, 0],
 [0, 3, 2, 1, 0, 0],
 [0, 1, 2, 3, 4, 0],
 [0, 0, 0, 0, 0, 0]]
```

$\circledast$

```
[[0, 1],
 [2, 3]]
```

$\longrightarrow$

```
[[ 3.,  5.,  8., 16.,  8.],
 [16., 29., 35., 42., 16.],
 [14., 18., 14., 10.,  0.],
 [ 6., 10., 14., 18.,  8.],
 [ 1.,  2.,  3.,  4.,  0.]]
```

Stride

- Stride is the number of steps the filter moves across the input matrix during the sliding window operation.
- When stride is 1:

```
[[1, 1, 2, 4],
 [5, 6, 7, 8],
 [3, 2, 1, 0],
 [1, 2, 3, 4]]
```

$\circledast$

```
[[0, 1],
 [2, 3]]
```

$\longrightarrow$

```
[[29., 35., 42.],
 [18., 14., 10.],
 [10., 14., 18.]]
```

- When stride is 2:

```
[[1, 1, 2, 4],
 [5, 6, 7, 8],
 [3, 2, 1, 0],
 [1, 2, 3, 4]]
```

$\circledast$

```
[[0, 1],
 [2, 3]]
```

$\longrightarrow$

```
[[29., 42.],
 [10., 18.]]
```

Channels (input)

- We usually have multiple ‘channels’. We use a 3D instead of 2D matrix.
- For example, a color image typically has 3 channels – Red, Green, and Blue.

Input data
(2,4,4)

```
[[[1, 1, 2, 4],
  [5, 6, 7, 8],
  [3, 2, 1, 0],
  [1, 2, 3, 4]],
 [[4, 3, 2, 1],
  [8, 7, 6, 5],
  [0, 1, 2, 3],
  [4, 3, 2, 1]]]
```

⊛

Filter
(2,2,2)

```
[[[0, 1],
  [2, 3]],
 [[1, 0],
  [3, 2]]]
```

→

(Output) feature map
(2, 2)

```
[[71., 72.],
 [28., 28.]]
```

We add the results of each channel

Channels (intermediate layers)

- Intermediate layers may have multiple channels as well.
- If we want multiple channels in the output, we can prepare **<u>multiple</u>** filters.

Input data
(2, 4, 4)

```
[[[1, 1, 2, 4],
  [5, 6, 7, 8],
  [3, 2, 1, 0],
  [1, 2, 3, 4]],
 [[4, 3, 2, 1],
  [8, 7, 6, 5],
  [0, 1, 2, 3],
  [4, 3, 2, 1]]]
```

⊛

Filter: (3, 2, 2)

```
[[[[0., 1.],
   [2., 3.]],
  [[1., 0.],
   [3., 2.]]],
 [[[2., 1.],
   [1., 2.]],
  [[2., 1.],
   [1., 2.]]],
 [[[1., 0.],
   [0., 1.]],
  [[1., 0.],
   [0., 1.]]]]
```

→

(Output) feature map
(3, 2, 2)

```
[[[71., 72.],
  [28., 28.]],
 [[53., 52.],
  [24., 24.]],
 [[18., 17.],
  [ 8.,  8.]]]
```

Pooling layer

- Pooling is a down-sampling operation (usually) along the spatial dimensions (width, height). Reduces the dimensionality of the input and helps to prevent overfitting. No new learnable parameters.
- Max/average pooling: takes the maximum/average of the input in a certain region.

Examples with filter size 2, stride 2:

```
[[3., 5., 8., 16.],
 [16., 29., 35., 42.],
 [14., 18., 14., 10.],
 [6., 10., 14., 18.]]
```

→

```
[[29., 42.],
 [18., 18.]]
```

Max pooling

```
[[3., 5., 8., 16.],
 [16., 29., 35., 42.],
 [14., 18., 14., 10.],
 [6., 10., 14., 18.]]
```

→

```
[[13.25, 25.25],
 [12.00, 14.00]]
```

Average pooling

Motivation of pooling layers

- Pooling makes it robust to small shifts and distortions in the input.

Examples where the input is shifted to the left:

```
[[3., 5., 8., 16.],
 [16., 29., 24., 42.],
 [14., 18., 14., 10.],
 [6., 10., 14., 18.]]
```

→

```
[[29., 42.],
 [18., 18.]]
```

```
[[5., 8., 16., 10.],
 [29., 24., 42., 3.],
 [18., 14., 10., 4.],
 [10., 14., 18., 5.]]
```

→

```
[[29., 42.],
 [18., 18.]]
```

Max pooling, filter size 2, stride 2

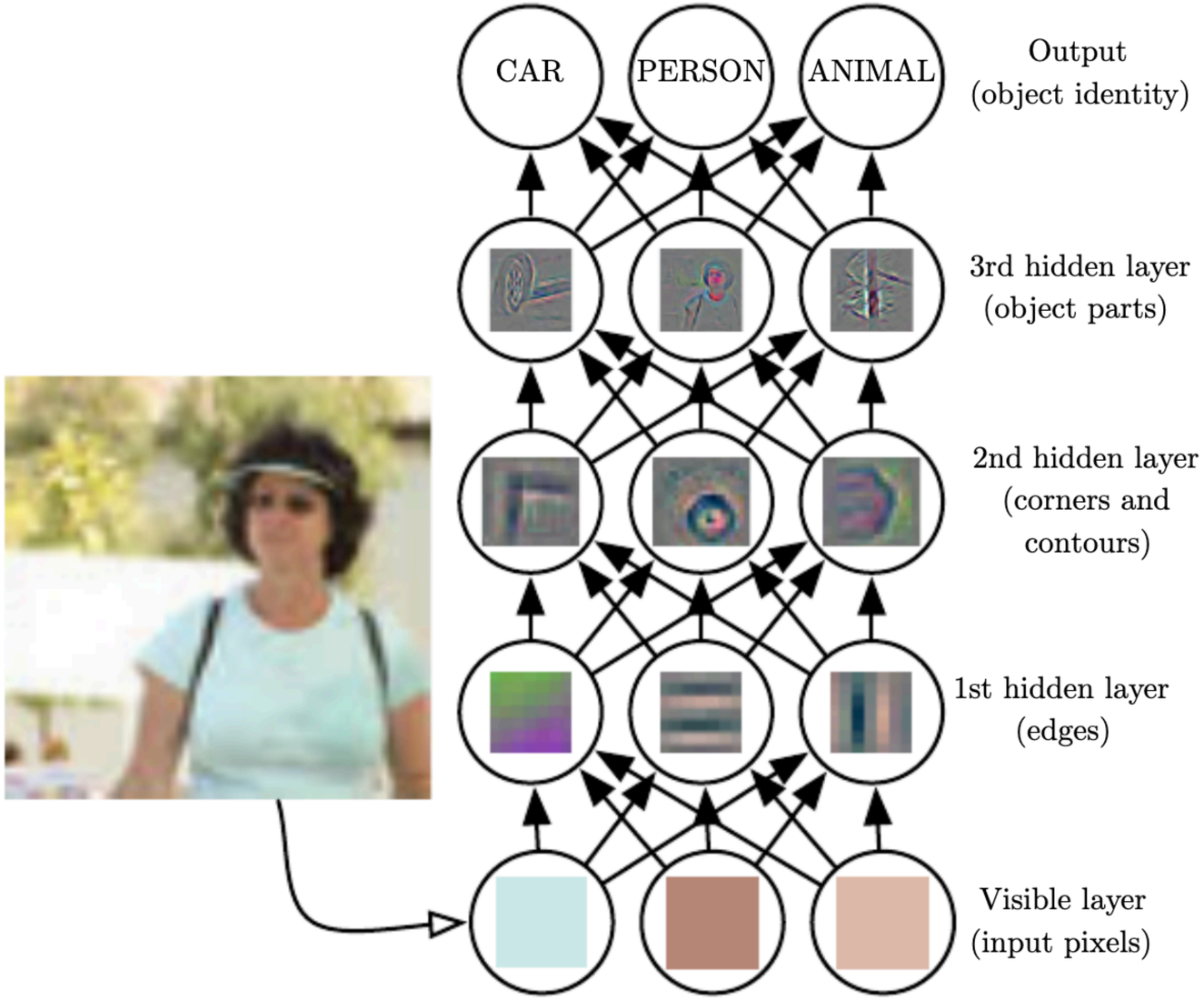
Simple convolutional neural net

```
import torch.nn as nn; import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5) # (in_cn, out_cn, kernel size)
        self.pool = nn.MaxPool2d(2, 2) # (kernel size, stride)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

code from: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Visualizing convolutional nets

- We can study convolution layers by reconstructing the input pixels that excite individual units in each layer, based on a deconvolutional net.
- Earlier layers are in charge on edges and corners, while later layers progressively identify more complex features.



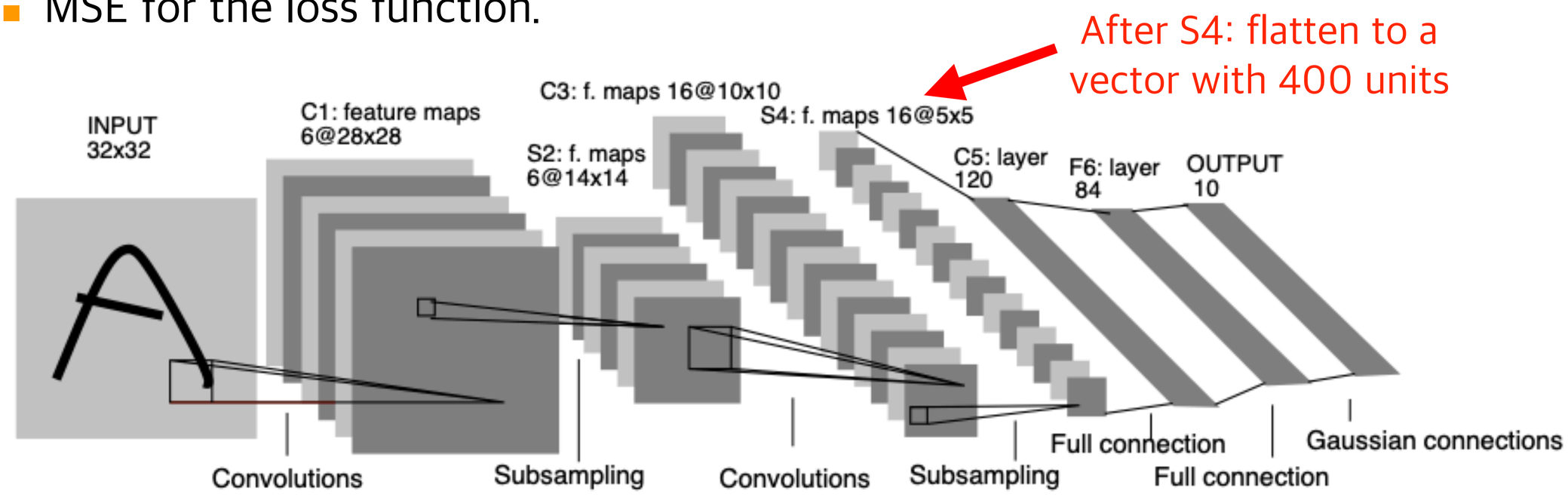
The figure is from Goodfellow et al. “Deep Learning” MIT Press 2016.

Visualization method proposed in Feiler & Fergus “Visualizing and Understanding Convolutional Networks” (ECCV 2014).

Contents

LeNet-5

- One of the very early CNNs proposed in 1998.
- Tanh activations at intermediate layers and RBF at output layer.
- MSE for the loss function.



- Exercise:
  - What is the filter size, padding, and stride in each convolution layer?
  - What is the window size for the pooling layers?

LeCun et al. Gradient-based learning applied to document recognition. IEEE Vol 86, Issue 11, 1998

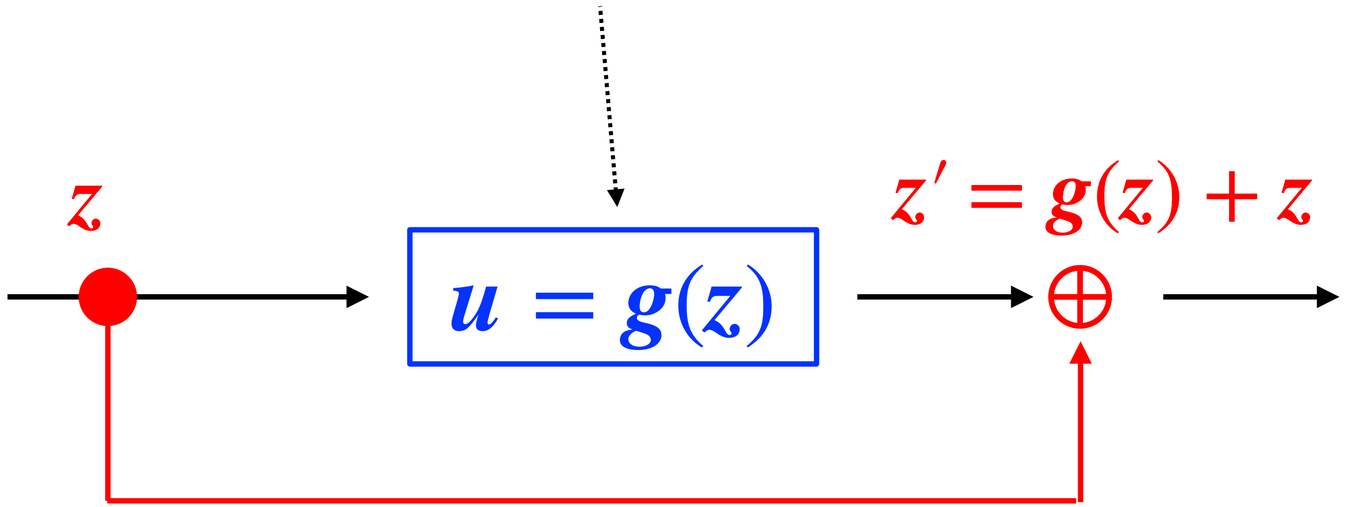
Solution

- Filter size is 5 x 5.
  - For 1st convolution: since 32 – f1 + 1 = 28, we can derive f1 = 5
  - For 2nd convolution: since 14 – f2 + 1 = 10, we can derive f2 = 5
  - No padding, stride is 1
- Window size of pooling operation is 2 x 2 with stride 2

Residual connections

- We feed the output of an earlier layer directly to a later layer and combine with the output of the layer one before.
- Since we skip one or more layers, we call this skip connection.
- An example of a **residual block**:

The rectangle is one or more layers with $\boldsymbol{u}$ as the output of the layer(s) and z is the input to the layer(s)



Global average pooling (GAP)

- Take the single average value of each channel of feature map and return a vector with length same as number of channels.
- Process to reduce 3D tensor (channel, height, width) to a 1D tensor (channel).

```
[[[1, 2],
  [3, 4]],
 [[5, 6],
  [7, 8]],
 [[9, 10],
  [11, 12]]]
```

(3, 2, 2)

flatten →

```
[1,
 2,
 3,
 .
 .
 .
 12]
```

(12, 1)

dense → [ ]

12 params

(1)

```
[[[1, 2],
  [3, 4]],
 [[5, 6],
  [7, 8]],
 [[9, 10],
  [11, 12]]]
```

(3, 2, 2)

GAP →

```
[ 2.5,
  6.5,
 10.5]
```

(3, 1)

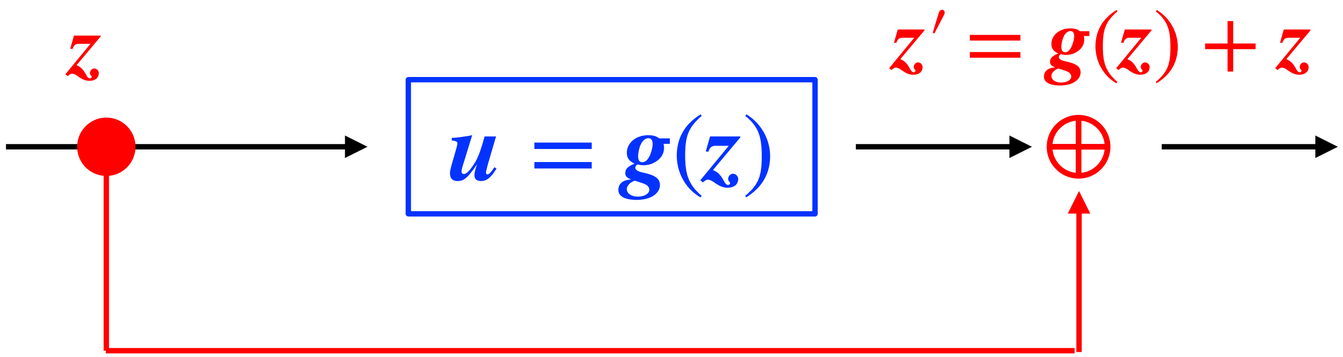
dense → [ ]

3 params

(1)

Residual connections

- From $z' = \boldsymbol{g}(z) + z$, we can derive: $\boldsymbol{g}(z) = z' - z$
- The layer that was bypassed can be expressed with the **residual**.
- Motivation: even if we suffer from some gradient issues in $\boldsymbol{g}(\,\cdot\,)$, e.g., gradient vanishing, we can propagate through the skip connection.

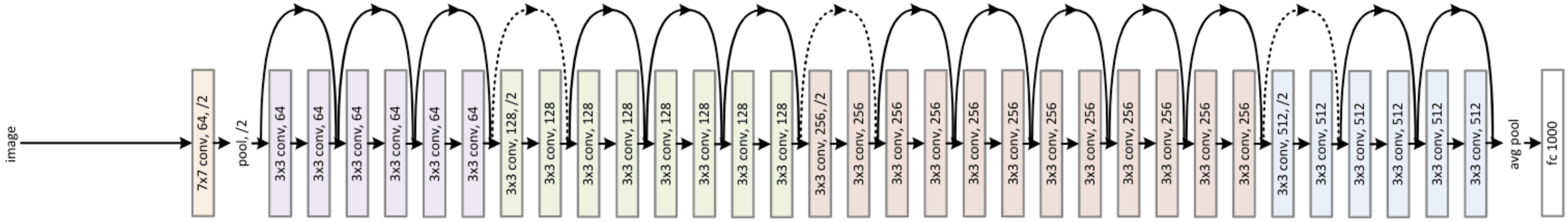


Residual nets (ResNets)

- Architecture proposed in 2016: ResNet-{18, 34, 101, 152}
- ResNet-34 architecture shown in the figure below.
- Successfully trained much deeper networks by utilizing batch norm, residual connections, and global average pooling (GAP).

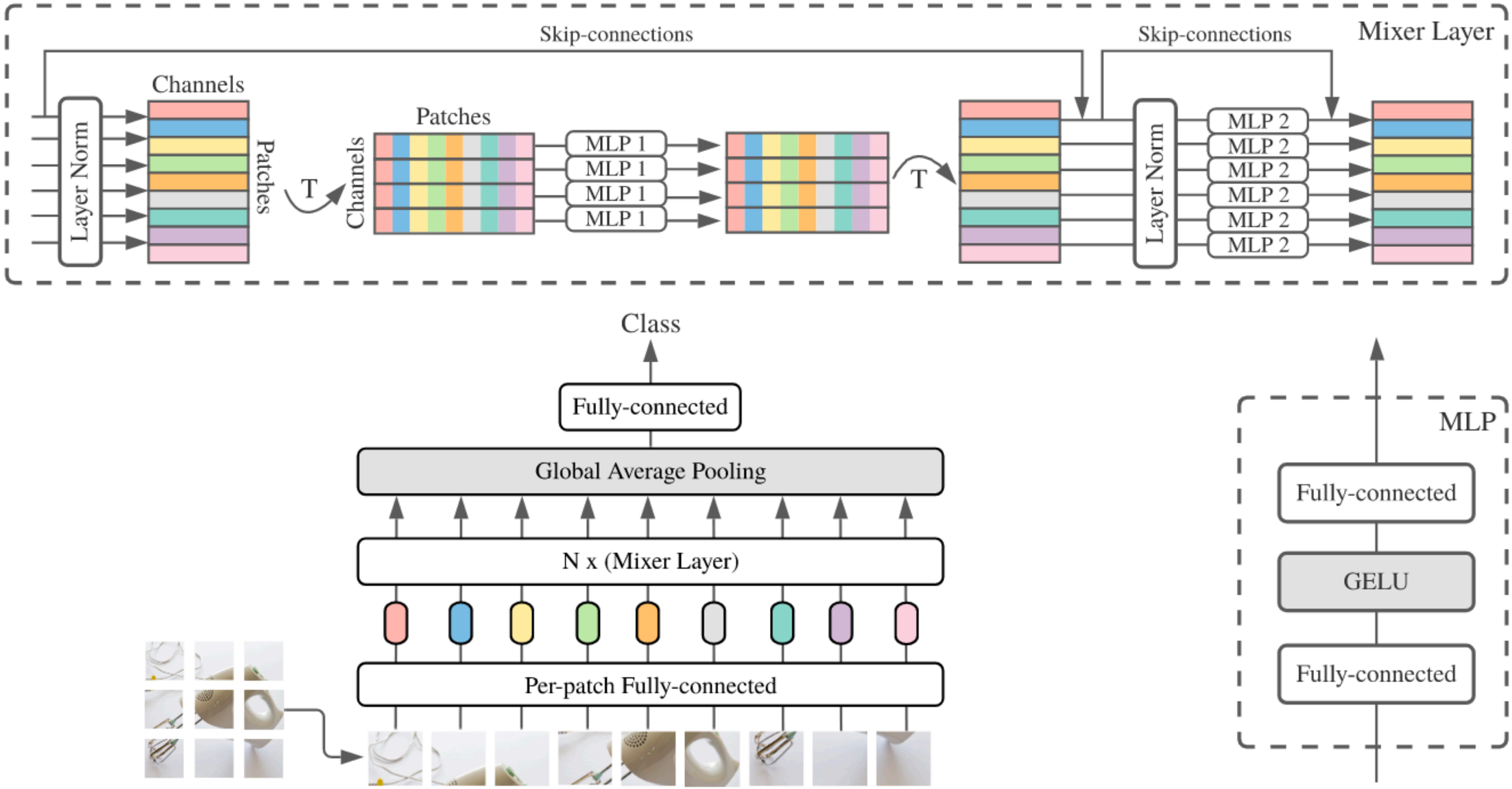
	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

Table 2. Top-1 error (%, 10-crop testing) on ImageNet validation. Here the ResNets have no extra parameter compared to their plain counterparts. Fig. 4 shows the training procedures.



He et al. Deep residual learning for image recognition. CVPR 2016.

MLP-Mixer



Tolstikhin et al. MLP-Mixer: An all-MLP Architecture for Vision. NeurIPS 2021.

Implementation of MLP-Mixer

- JAX/Flax implementation shown in the paper.

```
import einops
import flax.linen as nn
import jax.numpy as jnp

class MlpBlock(nn.Module):
  mlp_dim: int
  @nn.compact
  def __call__(self, x):
    y = nn.Dense(self.mlp_dim)(x)
    y = nn.gelu(y)
    return nn.Dense(x.shape[-1])(y)

class MixerBlock(nn.Module):
  tokens_mlp_dim: int
  channels_mlp_dim: int
  @nn.compact
  def __call__(self, x):
    y = nn.LayerNorm()(x)
    y = jnp.swapaxes(y, 1, 2)
    y = MlpBlock(self.tokens_mlp_dim, name='token_mixing')(y)
    y = jnp.swapaxes(y, 1, 2)
    x = x+y
    y = nn.LayerNorm()(x)
    return x+MlpBlock(self.channels_mlp_dim, name='channel_mixing')(y)

class MlpMixer(nn.Module):
  num_classes: int
  num_blocks: int
  patch_size: int
  hidden_dim: int
  tokens_mlp_dim: int
  channels_mlp_dim: int
  @nn.compact
  def __call__(self, x):
    s = self.patch_size
    x = nn.Conv(self.hidden_dim, (s,s), strides=(s,s), name='stem')(x)
    x = einops.rearrange(x, 'n h w c -> n (h w) c')
    for _ in range(self.num_blocks):
      x = MixerBlock(self.tokens_mlp_dim, self.channels_mlp_dim)(x)
    x = nn.LayerNorm(name='pre_head_layer_norm')(x)
    x = jnp.mean(x, axis=1)
    return nn.Dense(self.num_classes, name='head',
                    kernel_init=nn.initializers.zeros)(x)
```

MLP 1

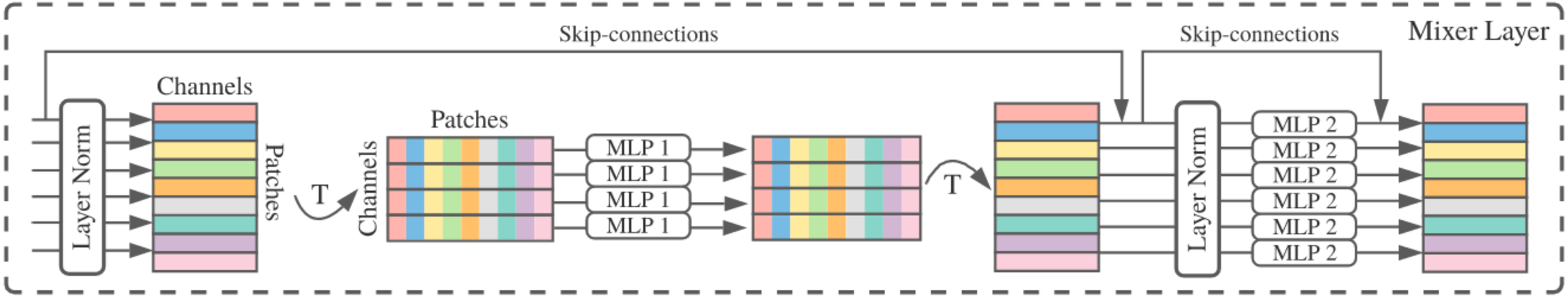
skip connections

MLP 2

skip connections

Interpretation of MLP-Mixer

- An extremely simple neural net based on MLPs without any convolutions can still perform close to state of the art.
  - (We haven't covered attention yet, but it is interesting that attention is not used.)
- Can be regarded as a very special CNN with:
  - 1x1 convolutions for channel mixing ↔ 1st FC layer in MLP2
  - Single-channel depth-wise convolutions of a full receptive field and parameter sharing for token mixing ↔ 1st FC layer in MLP1



1x1 convolutions for channel mixing ↔️ MLP2

1st FC layer in MLP2

1x1 convolution

```
import torch
channels, patches = 5, 9
weight = torch.randn(channels, channels)
bias = torch.randn(channels)
x = torch.randn(channels, patches)
linear = torch.nn.Linear(channels, channels)
conv = torch.nn.Conv1d(channels, channels, kernel_size=1)

# make sure the shape is the same:
print(linear.weight.shape, conv.weight.shape) # (5,5), (5,5,1)
print(linear.bias.shape, conv.bias.shape) # (5), (5)

# make sure conv/linear's weights/biases are the same:
conv.weight.data = weight.unsqueeze(-1)
linear.weight.data = weight
conv.bias.data = bias
linear.bias.data = bias

# check the results are the same for the 2 operations:
linear_results = linear(x).T
conv_results = conv(x)
print(torch.allclose(linear_results, conv_results)) # True!
```

Summary

- Initialization:
  - We need to be careful about how we initialize our networks.
  - Xavier's initialization and Kaiming's initialization are often used in practice.
- Normalization:
  - Batch norm and layer norm are popular techniques that makes learning easier with deeper networks.
  - We can be less careful about initialization with normalization.
- Convolutional neural nets:
  - Instead of MLPs, we consider convolutional layers with filters and pooling which may be more suitable for images.
  - Introduced a few more tools such as GAP and skip connections.
- Architectures:
  - LeNet-5 (classic), ResNet-{18, 34, 101, 152}, MLP-Mixer (modern)

Homework

- 1) [math] In batch norm, derive $\frac{\partial J}{\partial x_i}$ where x_i is the ith unit's pre-activation value and J is the loss/objective. Assume that we already know $\frac{\partial J}{\partial u_i}$, where u_i is the output after BN operation. Will be easier if you first derive $\frac{\partial J}{\partial \hat{x}_i}, \frac{\partial J}{\partial \sigma^2_{\mathcal{B}}}, \frac{\partial J}{\partial \mu_{\mathcal{B}}}$.
- 2) [architecture] In the slides today, we mostly ignored the bias parameters in convolutional layers. Do your research and find out the role of the bias parameter. Explain.
- 3) [programming (optional)] Read and understand the code that trains a one-hidden layer ReLU network with MNIST uploaded to ITC-LMS. Increase one more hidden layer with ReLU activation. Optionally try different initialization strategies. (You will need to modify the network architecture, backward pass, and optimization step. The point of this exercise is to not rely on loss.backward() and optimizer.step(). The code is based on PyTorch/Python but alternatives are fine.)

Schedule

- 04/11 Introduction
- 04/18 Regression 1: least squares regression
- 04/25 Regression 2: sparse learning & robust learning
- 05/09 Classification 1: least squares classification
- 05/16 Classification 2: support vector classification & probabilistic classification
- 05/23 Deep learning 1: MLPs, backprop, optimizers, regularizers
- 06/06 Deep learning 2: initialization, normalization, NNs for images
- 06/13 Deep learning 3: NNs for sequential data
- 06/20 Semi-supervised learning
- 06/27 Transfer learning
- 07/04 Dimensionality reduction: unsupervised algorithms
- 07/11 Dimensionality reduction: supervised algorithms
- 07/18 Advanced topics