Deep Learning and Optimization

Unpacking Transformers, LLMs and Diffusion

Session 3

olivier.koch@ensae.fr

Summary of Session 2

There is no reason for deep learning to work.

Inductive bias and the right loss function are key.

Optimization as a distance between two distributions (data and predictions).

Cross-entropy / negative-log-likelihood as a natural loss function.

We built a bigram model and a neural probabilistic model.

	Session	Date	Content
Foundations	1	Jan, 28	Intro to DL TP: micrograd
	2	Feb, 4	Fundamentals I: inductive bias, loss functions TP: bigram, MLP for next character prediction
	3	Feb, 11	Fundamentals II: DL architectures TP: tensor-based models
Applications	4	Feb, 18	Attention & Transformers TP: GPT from scratch
	5	Feb, 25	DL for Computer vision TP: convnets on CIFAR-10
	6	Mar, 11	VAE and Diffusion TP: diffusion from scratch Quiz / Exam

Let’s venture into the variations of a deep networks

Network architecture and inductive bias

Loss function

Activation function

Regularization

Initialization

Residual networks

Normalization

Dropout

Activation functions

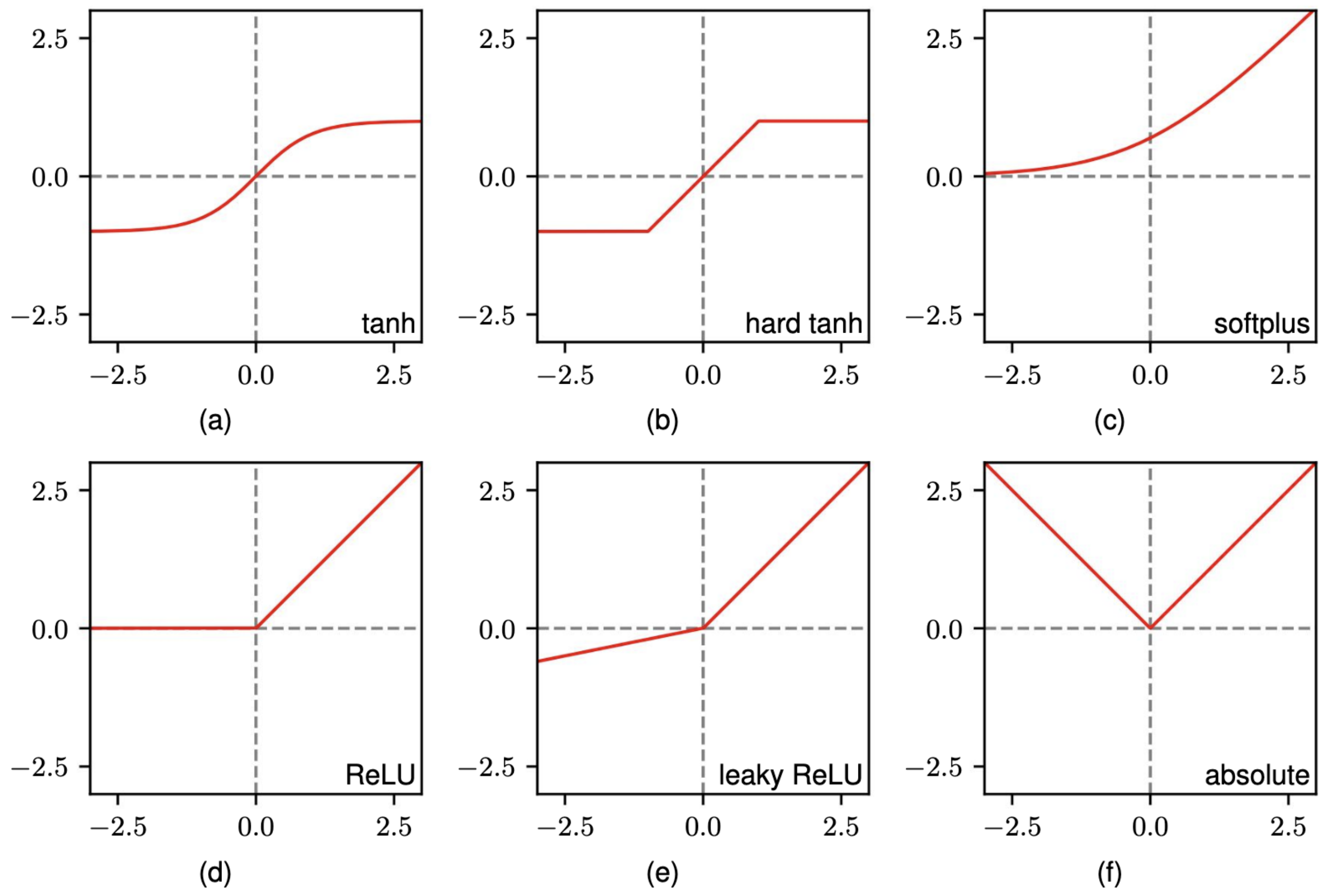


Figure 6.12 A variety of nonlinear activation functions.

Source: Deep Learning Foundations and Concept, Bishop, 2023

Activation functions

Only requirement: be differentiable.

Logistic and sigmoid → vanishing gradients ☹

ReLU gave a big improvement in training efficiency [1]

- Less sensitive to random initialization of the weights
- Well-suited for low-precision computation (8-bit vs 64-bit)
- Cheap to compute

→ By default, use ReLU or GeLU

[1] ImageNet Classification with Deep Convolutional Neural Networks, Krizhevsky, Sutskever and Hinton, NIPS 2012

GeLU (Gaussian Error Linear Units)

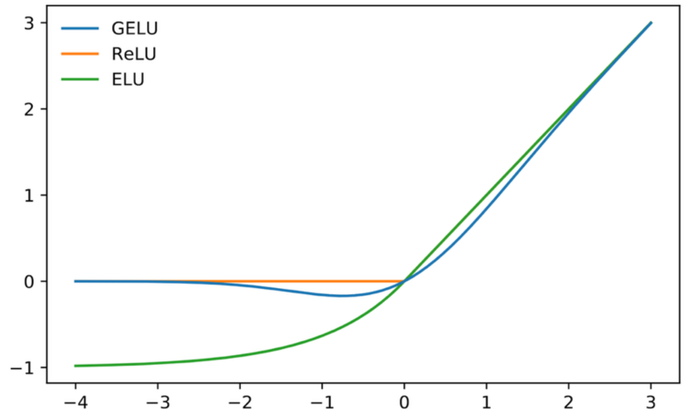
- Neuron inputs tend to follow a normal distribution, especially with Batch Normalization
- Inputs have a higher probability of being "dropped"

as x decreases

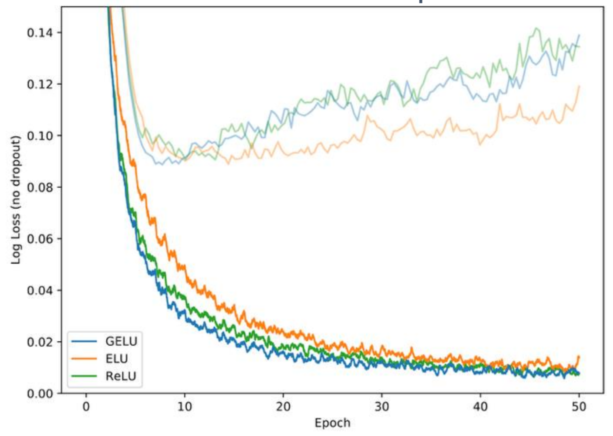
- Weight input by likelihood of input being greater than x

$$y = x * \Phi(x)$$

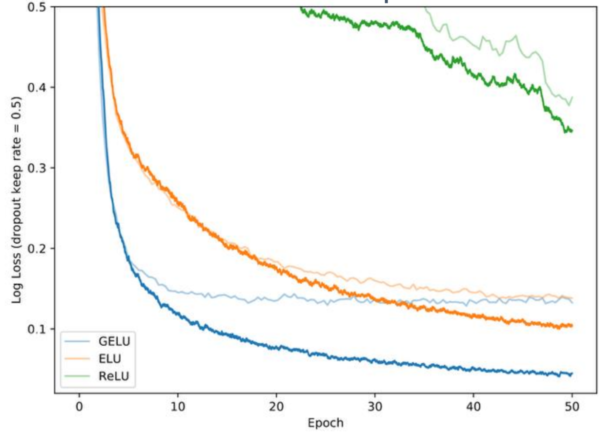
where $\Phi(x)$ is the cdf of the unit Normal distribution



MNIST w/ dropout



MNIST w/o dropout=0.5



[1] Gaussian Error Linear Units (GELUs), Dan Hendrycks, Kevin Gimpel, June 2023, arxiv

Let’s venture into the variations of a deep networks

Network architecture and inductive bias

Loss function

Activation function

Regularization

Initialization

Residual networks

Normalization

Dropout

Regularization

Regularization is about reducing the generalization gap between training and testing performance.

Implicit regularization is baked into SGD.

Explicit regularization is added through various methods (penalty term, data augmentation, dropout, etc.)

Regularization

Explicit regularization: adding a penalty term to the loss function

$$\hat{\omega} = \underset{\omega}{\operatorname{argmax}}\left[\prod_{i=1}^{N} Pr(y_i|x_i,\omega)\right]$$

$$\hat{\omega} = \underset{\omega}{\operatorname{argmax}}\left[\prod_{i=1}^{N} Pr(y_i|x_i,\omega)\, \mathrm{Pr}(\omega)\right]$$

$$\hat{\omega} = \underset{\omega}{\operatorname{argmin}}\left[-\sum_{i=1}^{N} \log[Pr(y_i|x_i,\omega)] + \log(\mathrm{Pr}(\omega))\right]$$

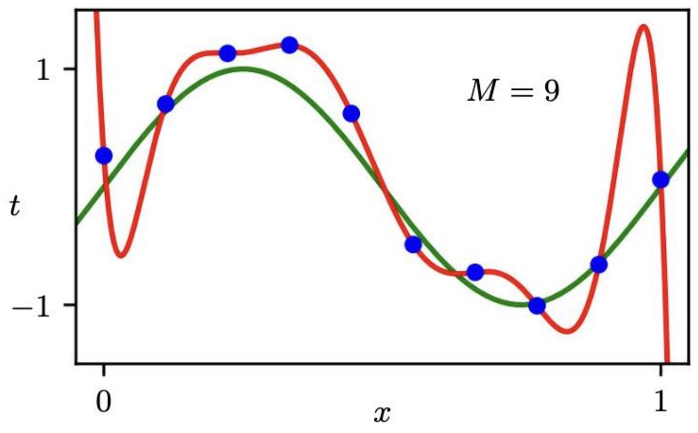
$$\hat{\omega} = \underset{\omega}{\operatorname{argmin}}\left[-\sum_{i=1}^{N} \ell_i[x_i,y_i] + \lambda\cdot \underset{\uparrow\ \text{penalty term}}{g}(\omega)\right]$$

Explicit regularization: adding a penalty term to the loss function

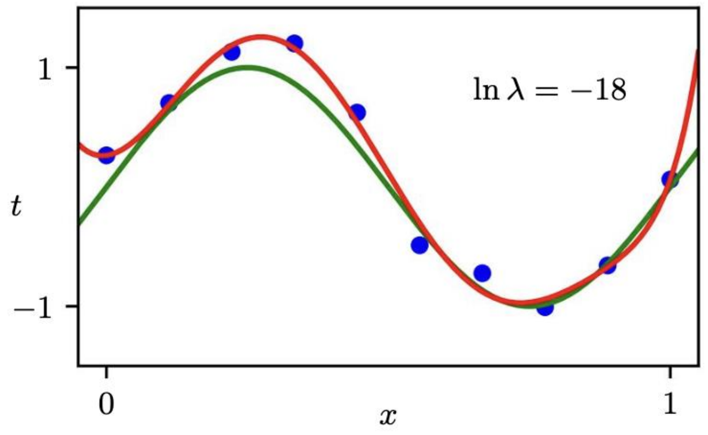
$$\hat{\omega} = \underset{\omega}{\operatorname{argmin}} \left[-\sum_{i=1}^{N} \ell_i[x_i, y_i] + \lambda \cdot \underbrace{\sum \omega_j^2}_{\text{L2 loss}} \right]$$

Regularization

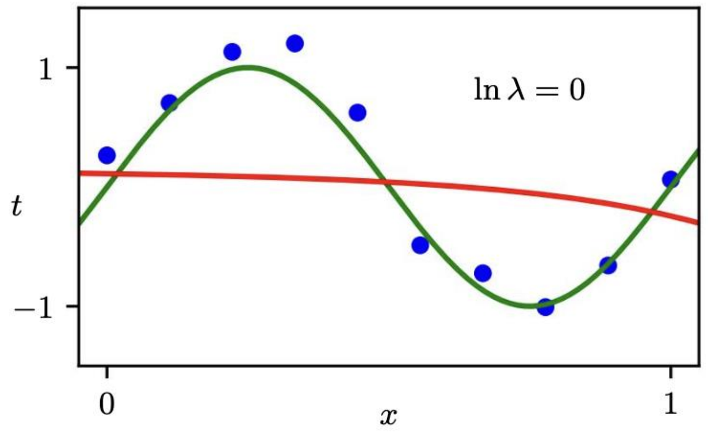
No regularization



Some regularization



A lot of regularization



Source: Deep Learning Foundations and Concept, Bishop, 2023

Regularization

Implicit regularization due to gradient descent

$$\omega_{t+1} = \omega_t - \eta \cdot \frac{\partial L}{\partial \omega}$$

$$L_{GD}[\omega] = L[\omega] + \frac{\eta}{4} \underbrace{\left\| \frac{\partial L}{\partial \omega} \right\|^2}_{\textit{implicit penalty}}$$

Implicit regularization due to stochastic gradient descent

If we denote L the average loss overall all samples and L_B the average loss over all batches:

$$L_{SGD}[\omega] = L_{GD}[\omega] + \frac{\eta}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_B}{\partial \omega} - \frac{\partial L}{\partial \omega}\right\|^2$$

$$L_{SGD}[\omega] = L[\omega] + \frac{\eta}{4}\left\|\frac{\partial L}{\partial \omega}\right\|^2 + \frac{\eta}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_B}{\partial \omega} - \frac{\partial L}{\partial \omega}\right\|^2$$

Larger steps regularize more

Smaller batches regularize more

Regularization

Smaller batches regularize more

Batch size (LR = 0.1)	Train error	Validation error
10	0.0%	37.6%
100	0.0%	43.2%
3000	36.0%	51.5%

Larger learning rates regularize more

LR (Batch size = 100)	Train error	Validation error
0.05	0.0%	44.6%
0.1	0.0%	43.2%
0.5	0.0%	40.9%

Example experiments on the MNIST-1D dataset with a 2-layer MLP

Source code: mnist1d.ipynb

Weight decay

Add a penalty term to the loss:

$$\mathcal{L}_{WD} = \mathcal{L} + \lambda \sum |w_j|^q$$

$q = 1$: Lasso → sparse model

$q = 2$: L_2-regularization → penalizes large magnitudes

Can be interpreted as the zero-mean Gaussian prior on the weights → inductive bias

Regularization

Techniques to add regularization:

- Early stopping
- Weight decay
- Ensembling
- Dropout
- Data augmentation
- Residual connections

Let’s venture into the variations of a deep networks

Network architecture and inductive bias

Loss function

Activation function

Regularization

Initialization

Residual networks

Normalization

Dropout

Initialization: forward pass

At each layer k, given weights $\boldsymbol{\Omega}$ with variance $\boldsymbol{\sigma}_{\boldsymbol{\Omega}}^{2}$ and pre-activations $\boldsymbol{f}_k$:

$$\boldsymbol{f}_k = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \cdot a[\boldsymbol{f}_{k-1}]$$

If $\boldsymbol{\sigma}_{\boldsymbol{\Omega}}^{2}$ is too large → exploding gradients

If $\boldsymbol{\sigma}_{\boldsymbol{\Omega}}^{2}$ is too small → vanishing gradients

Initialization: forward pass

At each layer k, given weights $\boldsymbol{\Omega}$ with variance $\boldsymbol{\sigma_{\Omega}^2}$ and pre-activations $\boldsymbol{f}_k$:

$$\boldsymbol{\sigma^2_{f_{k+1}} = \frac{1}{2} D_{h_k} \sigma^2_{\Omega} \sigma^2_{f_k}}$$

where $\boldsymbol{D_{h_k}}$ is the dimensionality of the input layer k.

Hence the optimal variance of the weights is:

$$\boldsymbol{\sigma^2_{\Omega} = \frac{2}{D_{h_k}}}$$

He initialization

Initialization: backward pass

Similarly, the optimal variance of the weights for the backward pass is:

$$\sigma_{\Omega}^{2} = \frac{2}{D_{h_{k+1}}}$$

Overall, the optimal variance of the weights is:

$$\sigma_{\Omega}^{2} = \frac{4}{D_{h_k} + D_{h_{k+1}}}$$

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, He et al., ICCV 2015

Let’s venture into the variations of a deep networks

Network architecture and inductive bias

Loss function

Activation function

Regularization

Initialization

Residual networks

Normalization

Dropout

Residual networks

More depth is not always better!

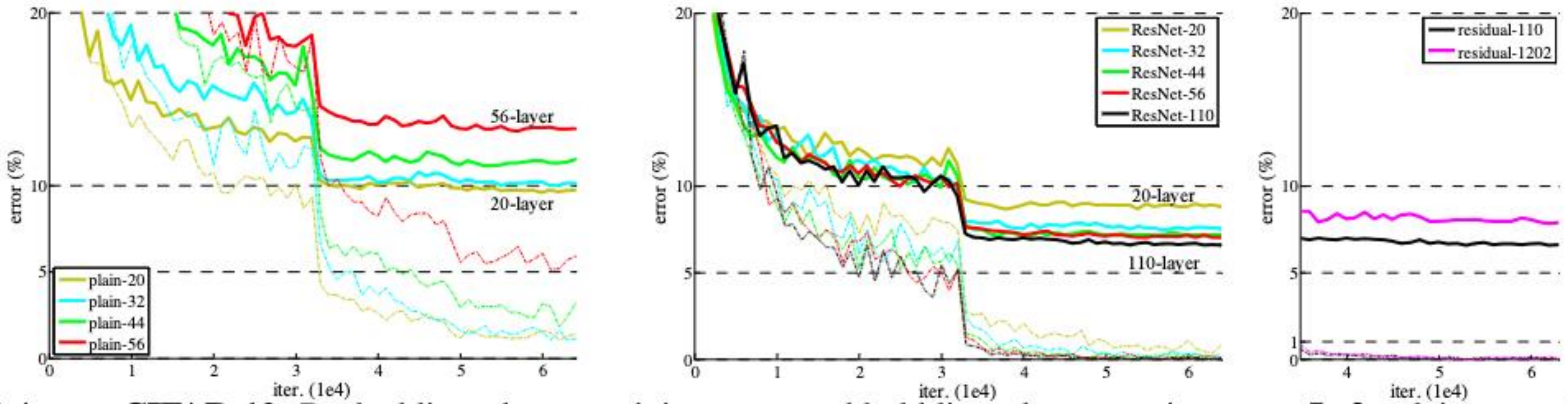


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

Deep Residual Learning for Image Recognition, Kaiming He et al., 2015

Residual networks

Gradient descent assumes that the function is smooth.

Unfortunately, the loss becomes less and less smooth with more depth (shattered gradients).

Residual networks

Shattered gradients

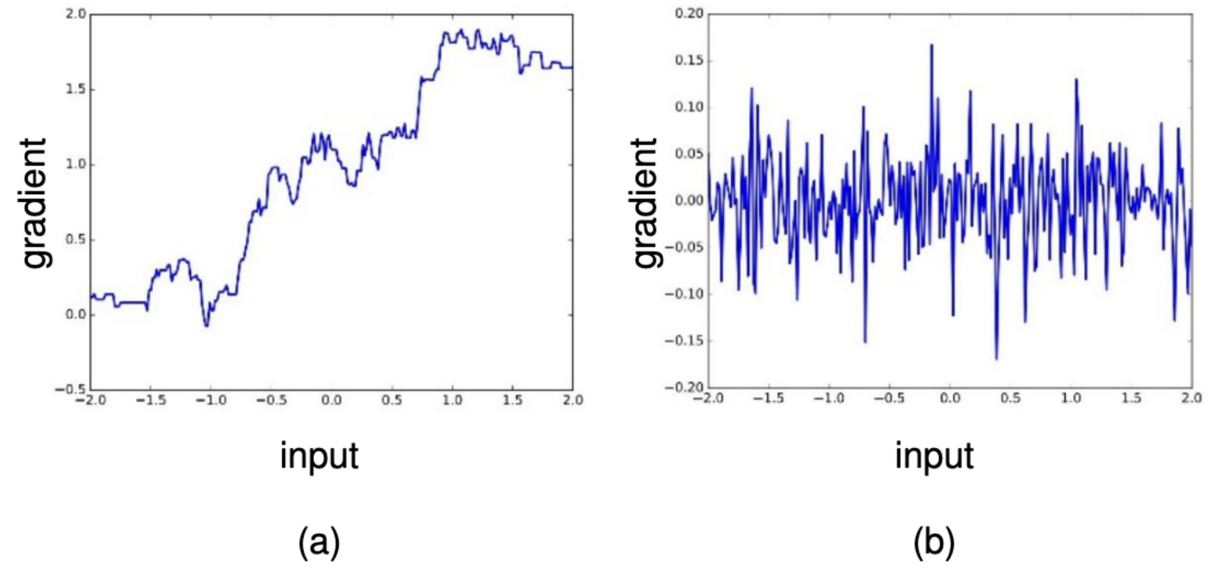


Figure 9.12 Plots of the Jacobian for networks with a single input and a single output, showing (a) a network with two layers of weights, (b) a network with 25 layers of weights

Residual networks

This can be addressed with skip connections.

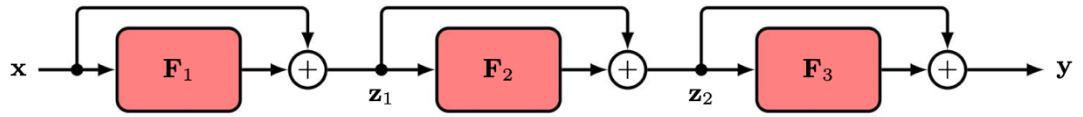


Figure 9.13 A residual network consisting of three residual blocks, corresponding to the sequence of transformations (9.35) to (9.37).

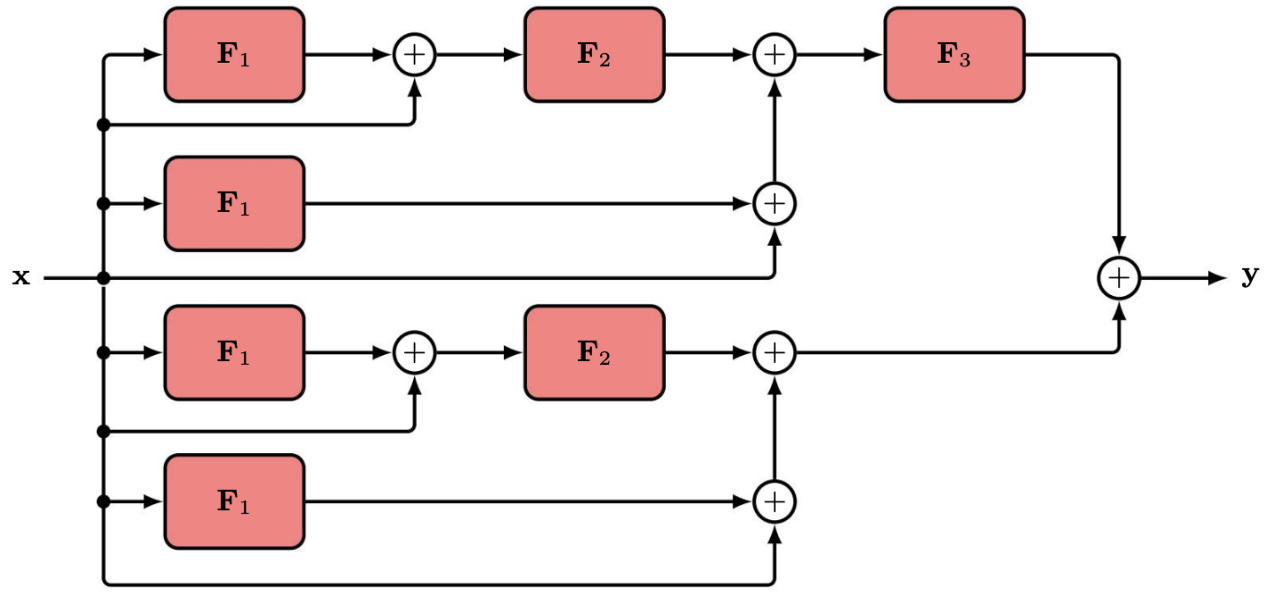


Figure 9.15 The same network as in Figure 9.13, shown here in expanded form.

Residual networks

This can be addressed with skip connections.

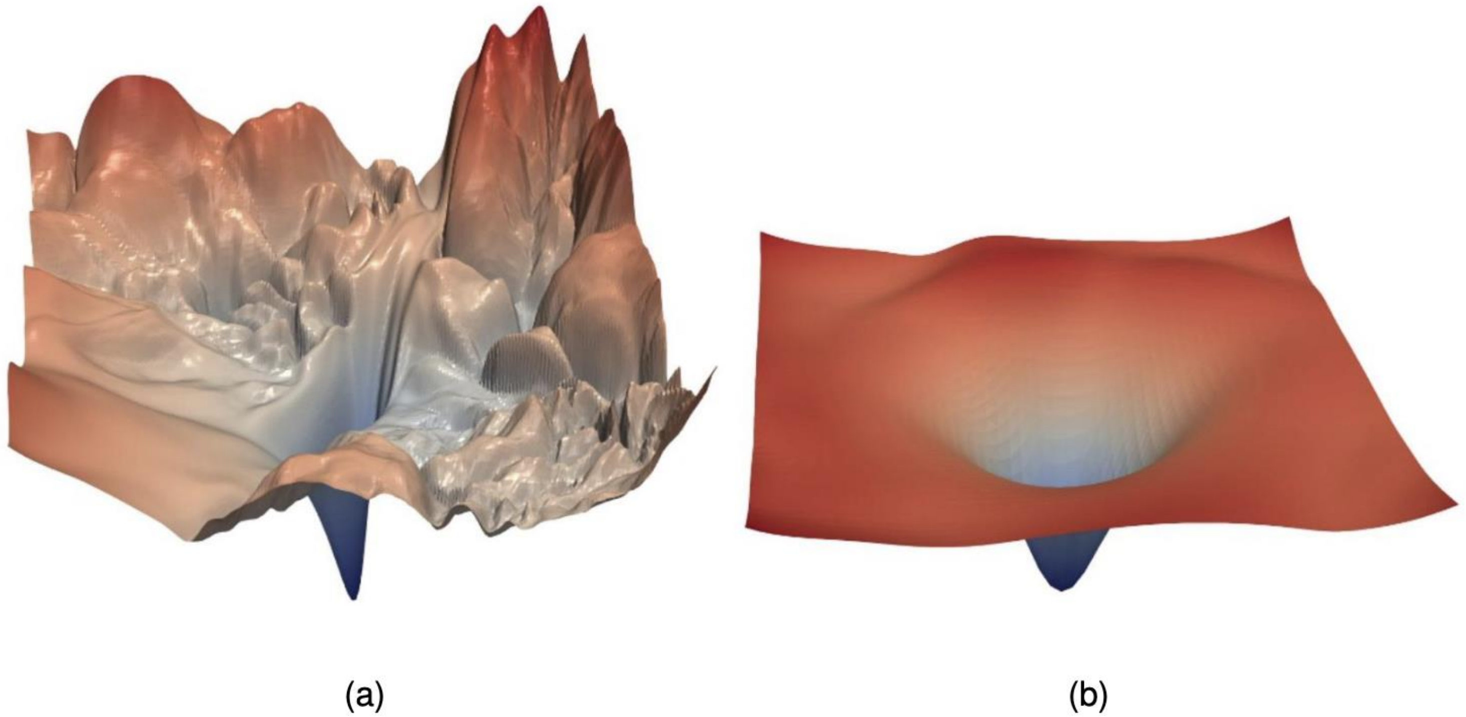


Figure 9.14 (a) A visualization of the error surface for a network with 56 layers. (b) The same network with the inclusion of residual connections, showing the smoothing effect that comes from the residual connections. [From Li *et al.* (2017) with permission.]

Source: Deep Learning Foundations and Concept, Bishop, 2023

Residual networks

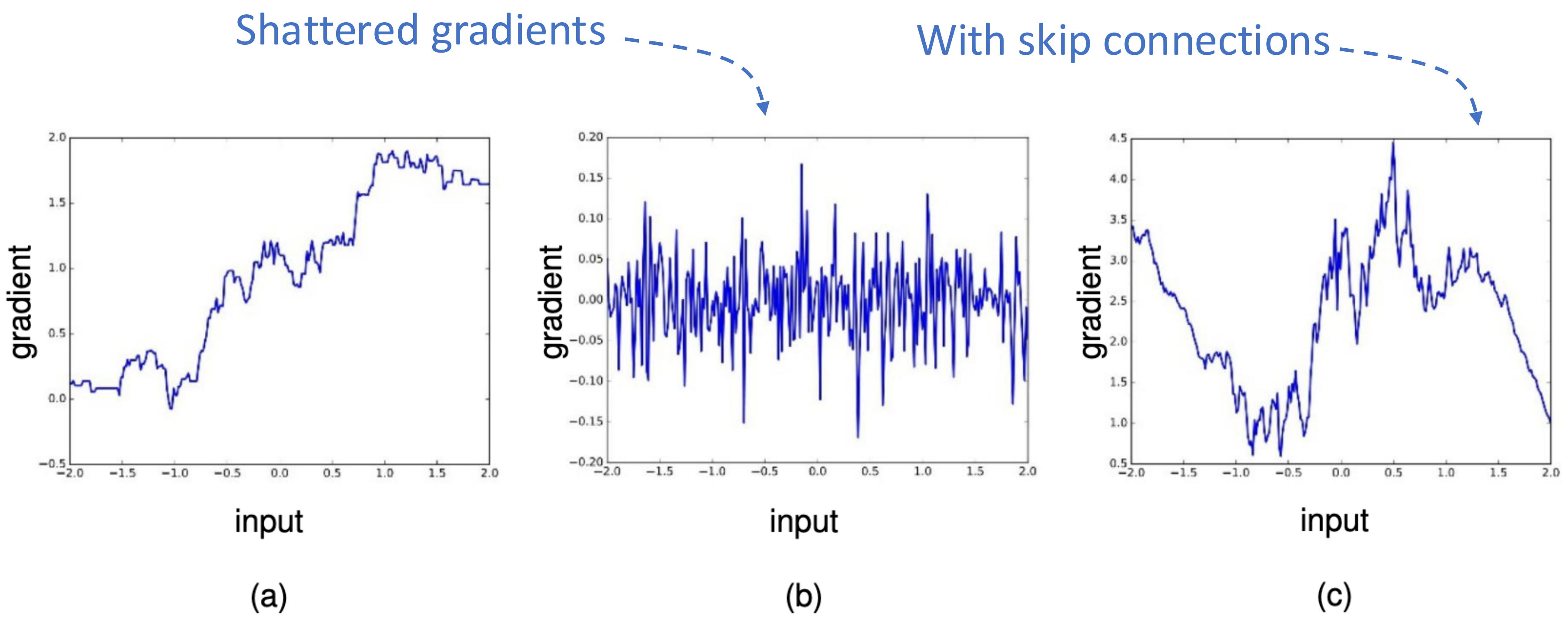
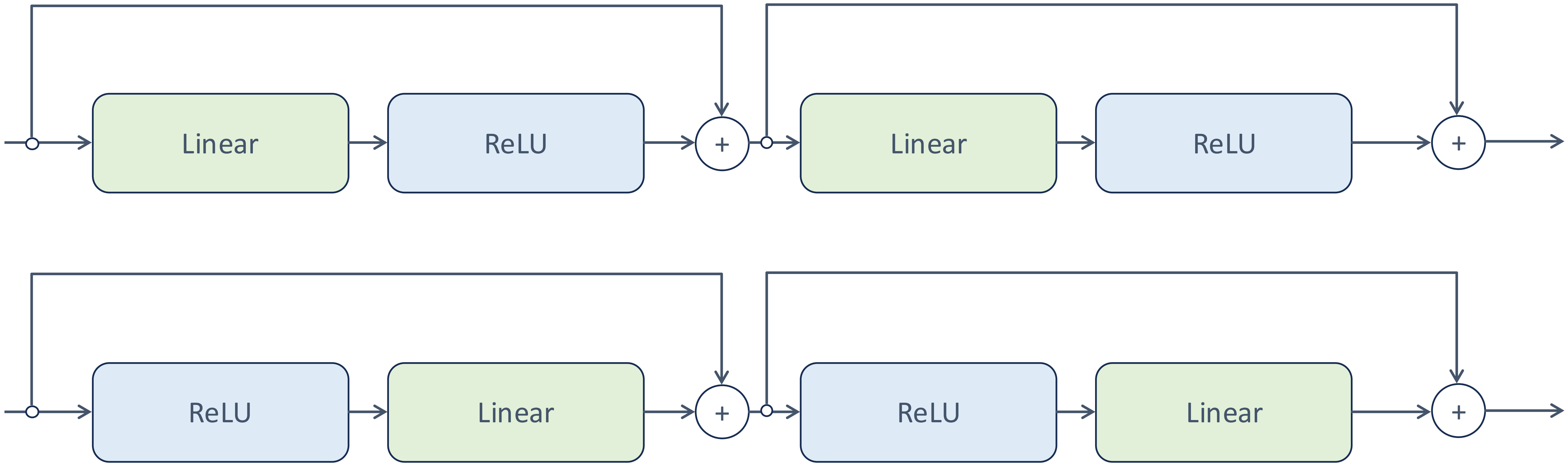


Figure 9.12 Plots of the Jacobian for networks with a single input and a single output, showing (a) a network with two layers of weights, (b) a network with 25 layers of weights, and (c) a network with 51 layers of weights together with residual connections. [From Balduzzi *et al.* (2017) with permission.]

Source: Deep Learning Foundations and Concept, Bishop, 2023

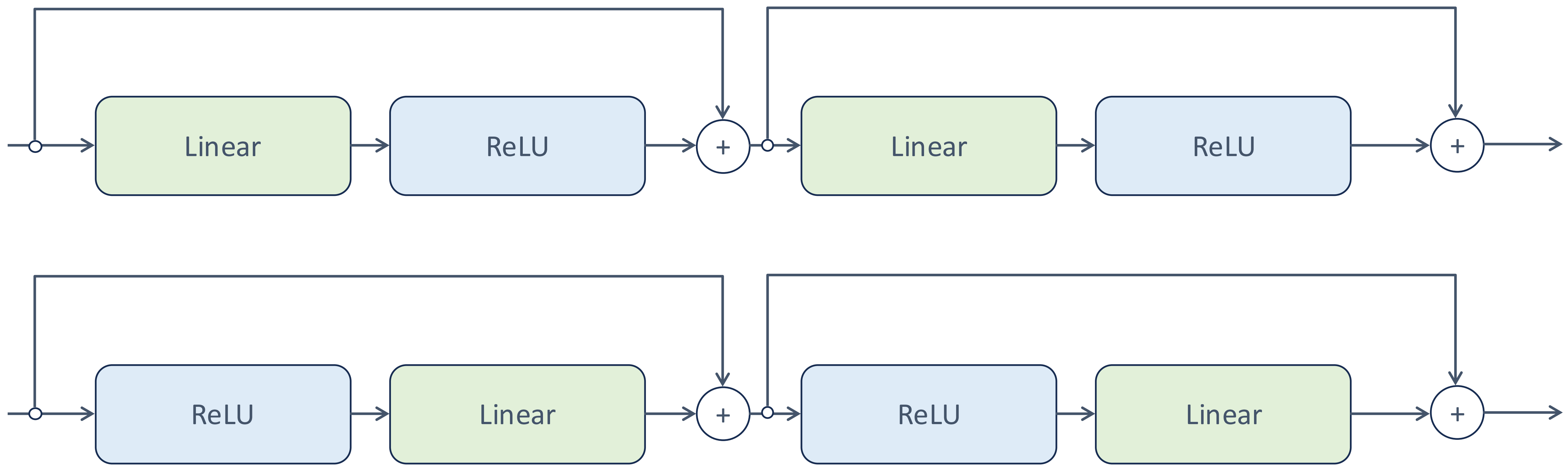
Residual networks

Two alternative ways of using residual connections.



Residual networks

Either way, variance doubles after a residual connection → batch norm!



Let’s venture into the variations of a deep networks

Network architecture and inductive bias

Loss function

Activation function

Regularization

Initialization

Residual networks

Normalization

Dropout

Batch norm

Batch Norm addresses the exploding gradient problem.

Alternative intuition:

We initialize the weights so that they have a nice distribution.

Why don’t we do this at each pass then? ☺

That’s Batch Norm!

Introduced by Ioff & Szegedy (2015)

Batch norm

Shift and scale each activation so that their mean and variance across the batch become values that are learned during training.

Not constant values!

Batch norm

Compute m_B and s_B (mean and variance) over the batch during training

Normalize activations $h_i = \frac{h_i - m_B}{s_B + \epsilon}$

Scale and shift: $h_i = \gamma \cdot h_i + \delta$

γ and δ are learned during training, for each hidden unit (not each layer)

For K layers containing each D units, that's $2 * K * D$ extra parameters.

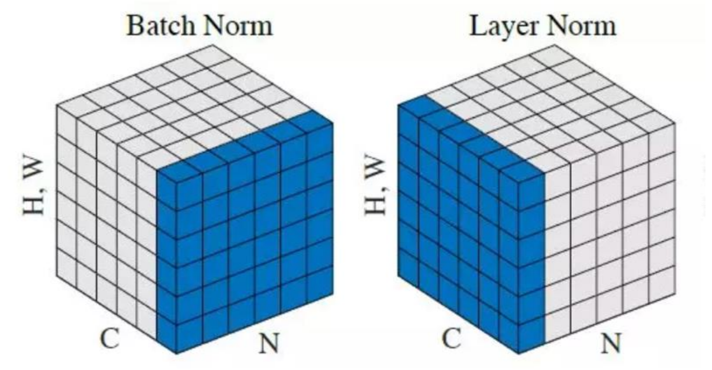
Batch norm

Downsides of Batch Norm

- Prone to bugs (need to freeze during inference)
- More parameters to learn
- Introduces dependencies between the training samples
- Needs to recompute the statistics on the whole dataset at testing time

Layer norm

Compute the layer normalization statistics over all the hidden units in the same layer.

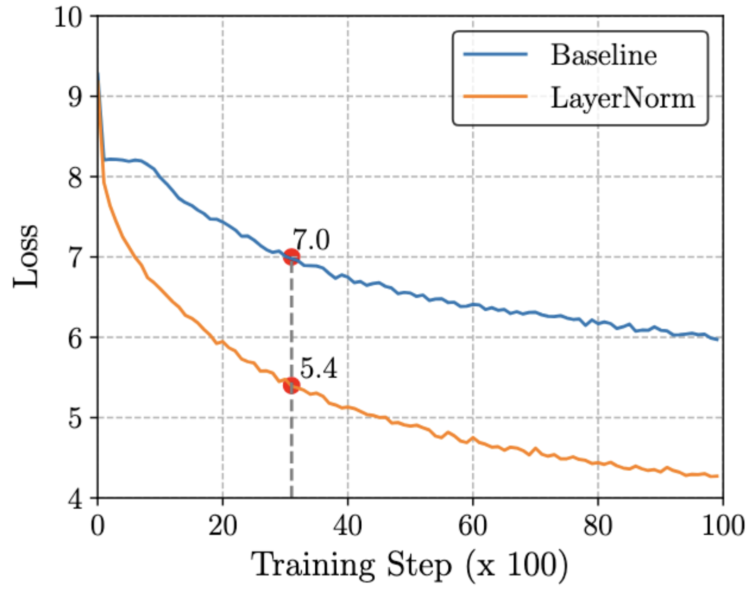


All the hidden units in a layer share the same normalization terms μ and σ, but different training samples have different normalization terms.

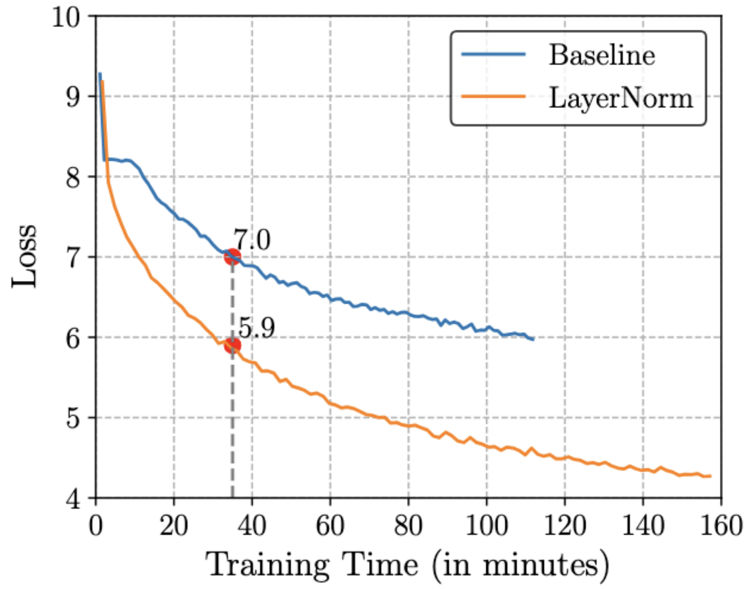
Layer Normalization, Ba, Kiros, Hinton, 2016

Layer norm

Normalization helps your network converge *faster*



(a) Training loss vs. training steps.



(b) Training loss vs. training time.

[1] Root Mean Square Layer Normalization, B. Zhang, R. Sennrich, NeurIPS 2019, arxiv

RMS Norm

LayerNorm is expensive. Center the variance, not the mean.

Learn the mean!

$$\bar{a}_i = \frac{a_i}{\mathrm{RMS}(\mathbf{a})} g_i, \quad \text{where } \mathrm{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}.$$

learned parameter

[1] Root Mean Square Layer Normalization, B. Zhang, R. Sennrich, NeurIPS 2019, arxiv

RMS Norm

LayerNorm is expensive. Center the variance, not the mean.

Learn the mean!

$$\bar{a}_i = \frac{a_i}{\mathrm{RMS}(\mathbf{a})} g_i, \quad \text{where } \mathrm{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} a_i^2}.$$

Applied on the last dimension!

[1] Root Mean Square Layer Normalization, B. Zhang, R. Sennrich, NeurIPS 2019, arxiv

RMS Norm

LayerNorm is expensive. Center the variance, not the mean.

Learn the mean!

$$\bar{a}_i = \frac{a_i}{\mathrm{RMS}(\mathbf{a})} g_i, \quad \text{where } \mathrm{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}.$$

X = [batch_size, seq_len, hidden_dim]

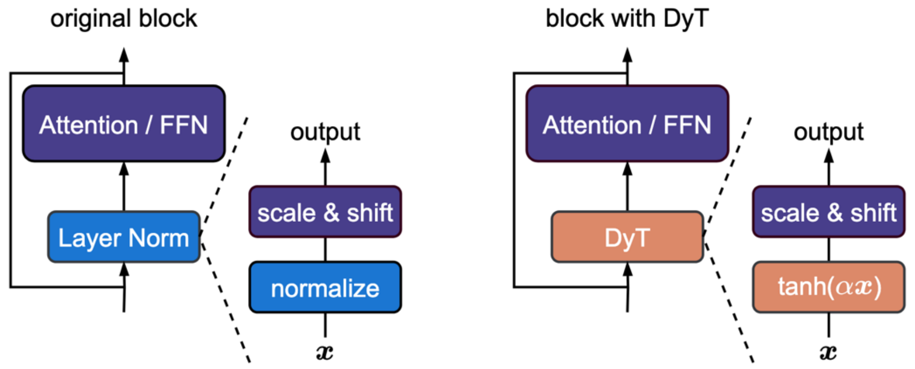
➔ Normalize across hidden_dim

➔ Compute batch_size x seq_len RMS values

[1] Root Mean Square Layer Normalization, B. Zhang, R. Sennrich, NeurIPS 2019, arxiv

DyT (Dynamic Tanh)

- Layer Norm produces tanh-like input-output mapping
- Replace normalization by a parameter in the activation function



score / loss	RMSNorm	DyT	change
LLaMA 7B	0.513 / 1.59	0.513 / 1.60	- / ↑0.01
LLaMA 13B	0.529 / 1.53	0.529 / 1.54	- / ↑0.01
LLaMA 34B	0.536 / 1.50	0.536 / 1.50	- / -
LLaMA 70B	0.549 / 1.45	0.549 / 1.45	- / -

[1] Transformers without Normalization, J. Zhu, X. Chen, K. He, Y. LeCun, Z. Liu, CVPR 2025, arxiv

Let’s venture into the variations of a deep networks

Network architecture and inductive bias

Loss function

Activation function

Regularization

Initialization

Residual networks

Batch norm, layer norm

Dropout

Dropout

Delete neurons at random during training -- not during inference!

Implicit way of averaging many models at once

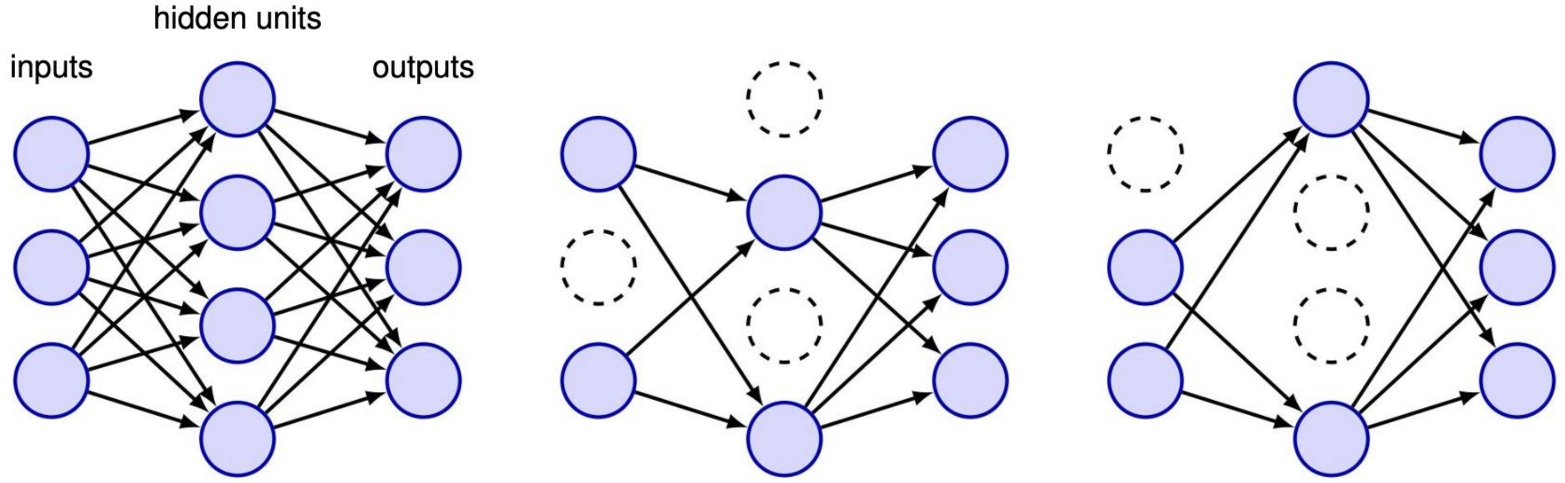


Figure 9.17 A neural network on the left along with two examples of pruned networks in which a random subset of nodes have been omitted.

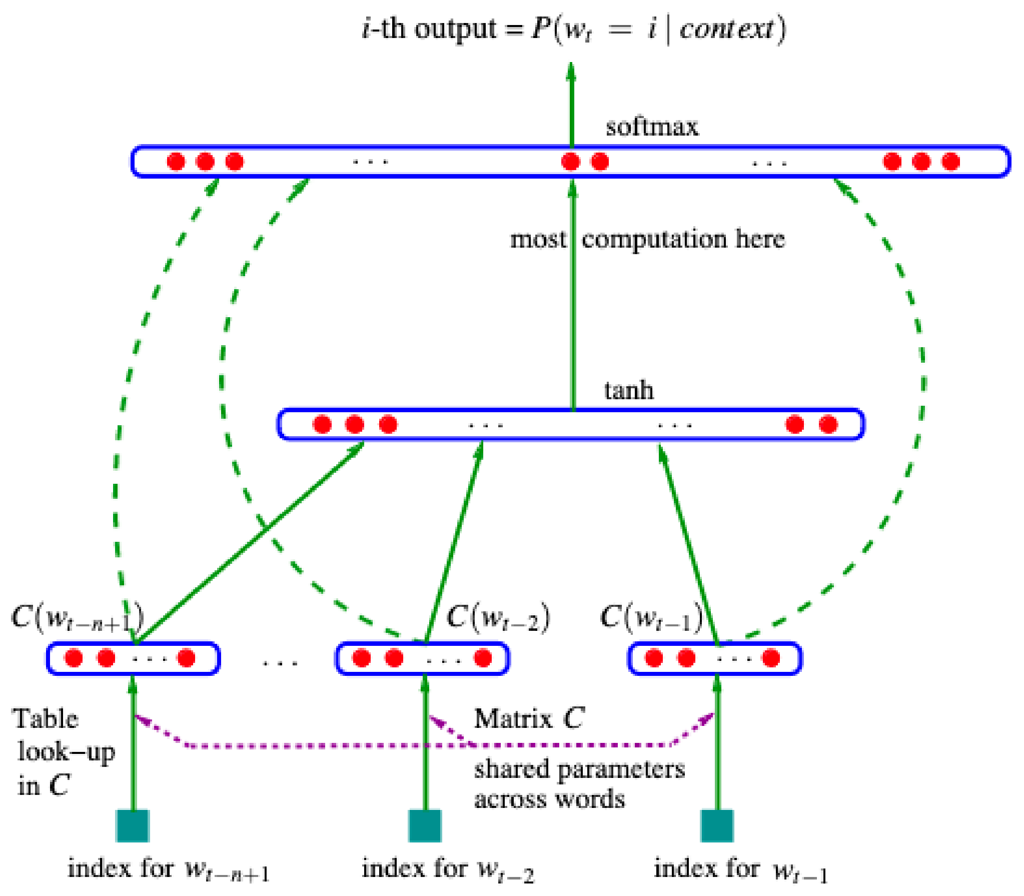
Summary

Key ingredients to make deep learning work:

- Activation function
- Regularization
- Initialization
- Residual networks
- Normalization
- Dropout

Going beyond N-gram with Recurrent Neural Networks (RNNs)

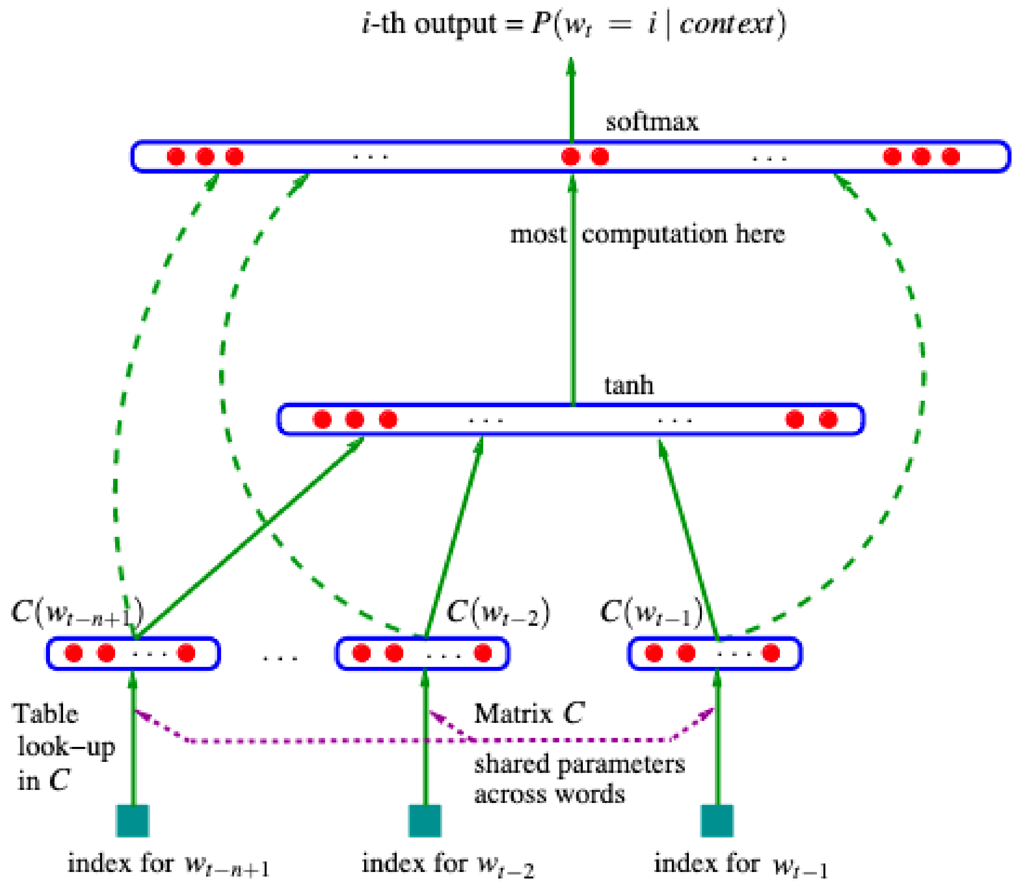
Neural Probabilistic Language Model



A Neural Probabilistic Language Model, Bengio et al, 2003

Going beyond N-gram with Recurrent Neural Networks (RNNs)

Neural Probabilistic Language Model



Main limitation:
does not model temporal dependencies

A Neural Probabilistic Language Model, Bengio et al, 2003

Going beyond N-gram with Recurrent Neural Networks (RNNs)

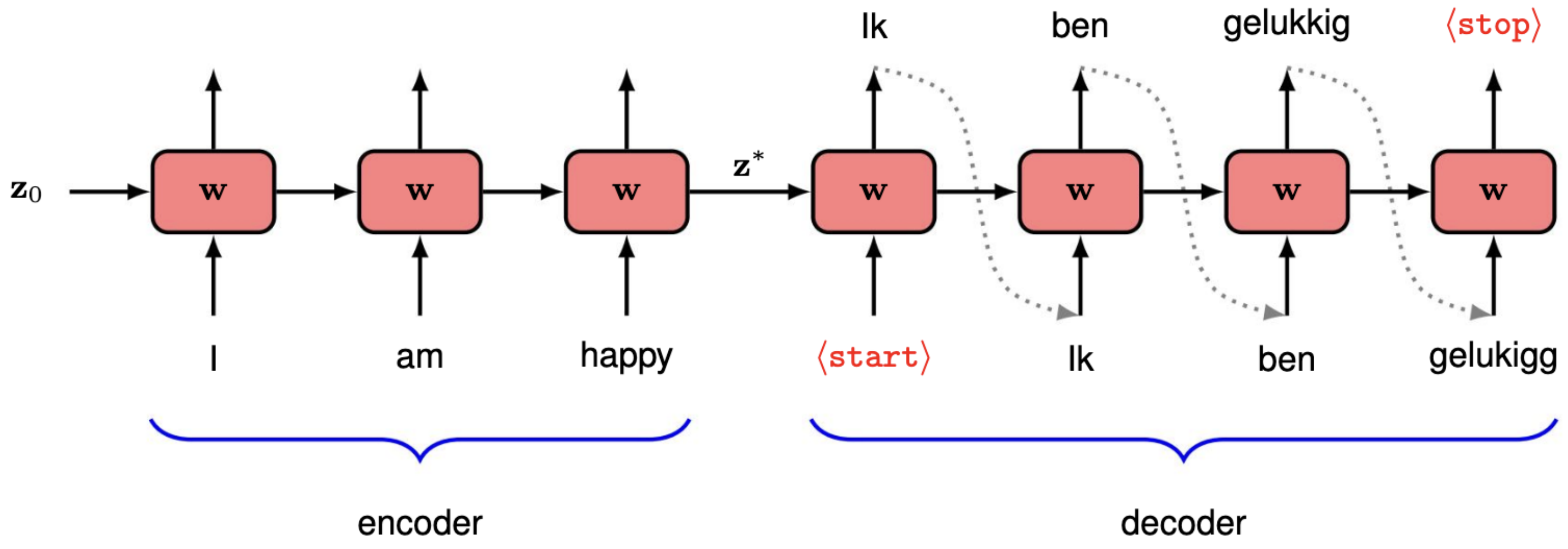


Figure 12.14 An example of a recurrent neural network used for language translation. See the text for details.

Going beyond N-gram with Recurrent Neural Networks (RNNs)

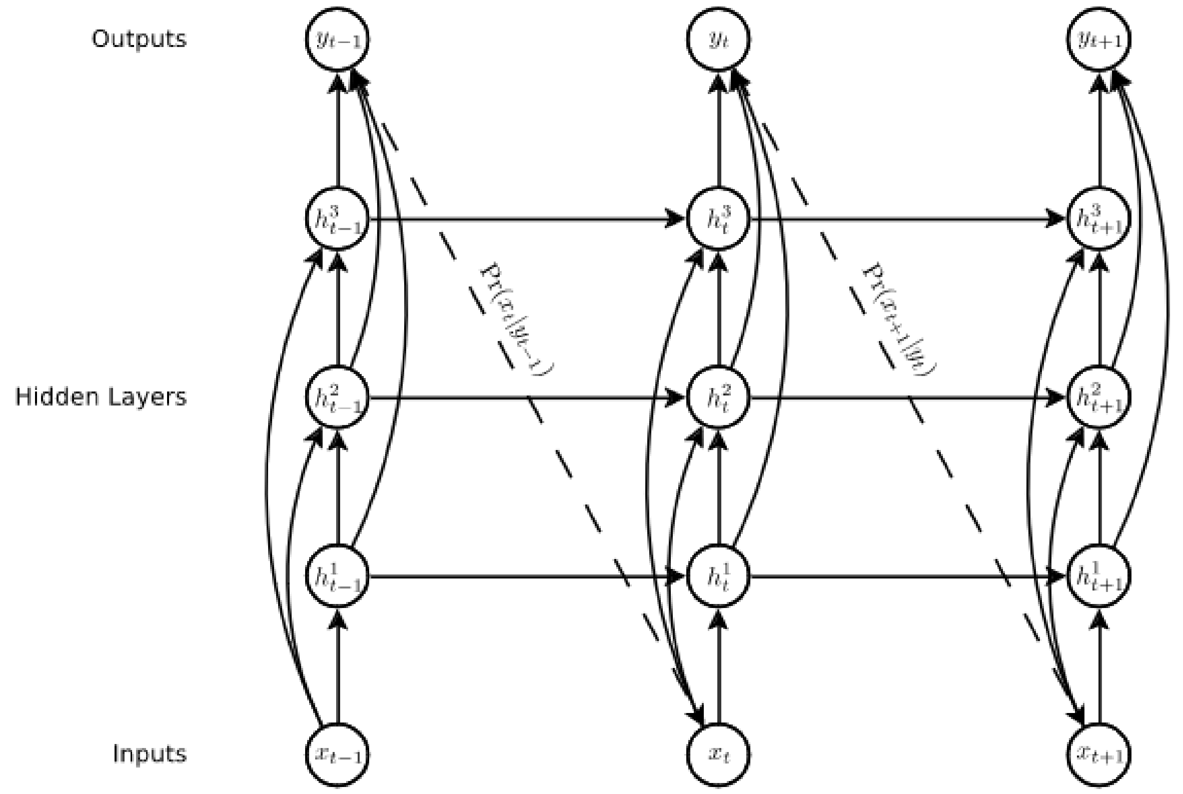


Figure 1: **Deep recurrent neural network prediction architecture.** The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions.

Generating Sequences With Recurrent Neural Networks, A. Graves, 2014

Going beyond N-gram with Recurrent Neural Networks (RNNs)

Standard RNNs are unable to store information about past inputs for very long.

Learning Long-Term Dependencies with Gradient Descent is Difficult, Bengio et al, 1994.

Solution: LSTM

Long Short-term Memory is an RNN architecture designed to be better at storing and accessing information than standard RNNs.

Generating Sequences With Recurrent Neural Networks, A. Graves, 2014

Going beyond N-gram with Recurrent Neural Networks (RNNs)

Better LSTMs

Sequence to sequence learning with neural networks, Sutskever, I., Vinyals, O., and Le, Q. , NIPS 2014.

Neural Machine Translation by Jointly Learning to Align and Translate, D. Bahdanau, K. Cho, Y. Bengio, 2015

GRUs (Gated Recurrent Units)

On the Properties of Neural Machine Translation: Encoder-Decoder Approaches, K. Cho et al, 2014

GRU and LSTM are comparable in performance

Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, J. Chung, C. Gulcehre, K. Cho, Y. Bengio, 2014.

Fundamental limitations of recurrent networks

Struggle to learn long-term dependencies (vanishing gradients)

Slow/inefficient training (sequential, not parallel, processing)

Difficulty handling variable-length sequences efficiently

Attention and Transformers

The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and the service.

Attention and Transformers

The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and the service.

Practical 3

Let’s learn tensor-based deep learning and apply it to MNIST.

Practical 3

1. Build an MLP from scratch using tensors
2. Apply it to MNIST (hand-written digits)
3. Sanity-check against pytorch .forward()
4. Unlock nn.Linear()

Practical 3

In Practical 1, you build backprop by hand

```
class Value:
 def __init__(self, data, _children=(), _op='', label=’’):
  self.data = data
  self.grad = 0.0
  self._backward = lambda: None
  self._prev = set(_children)
  self._op = _op
  self.label = label

 def __repr__(self):
  return f"Value(data={self.data})"

 def __add__(self, other): # TODO: ex.1
  other = other if isinstance(other, Value) else Value(other)
  out = Value(self.data + other.data, (self, other), ‘+’)

  def _backward(): # ex. 4
   self.grad += 1.0 * out.grad
   other.grad += 1.0 * out.grad
   out._backward = _backward

  return out
```

Practical 3

In Practical 1, you build backprop by hand

```
class Value:
 def __init__(self, data, _children=(), _op='', label=’’):
  self.data = data
  self.grad = 0.0
  self._backward = lambda: None
  self._prev = set(_children)
  self._op = _op
  self.label = label

 def __repr__(self):
  return f"Value(data={self.data})"

 def __add__(self, other): # TODO: ex.1
  other = other if isinstance(other, Value) else Value(other)
  out = Value(self.data + other.data, (self, other), ‘+’)

  def _backward(): # ex. 4
   self.grad += 1.0 * out.grad
   other.grad += 1.0 * out.grad
   out._backward = _backward

  return out
```

Does not work like this in the real world!

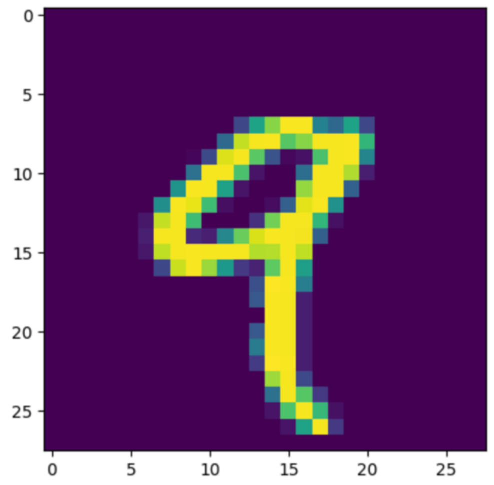
Deep learning is about tensor computation!

Practical 3

Typical linear transformation with pytorch

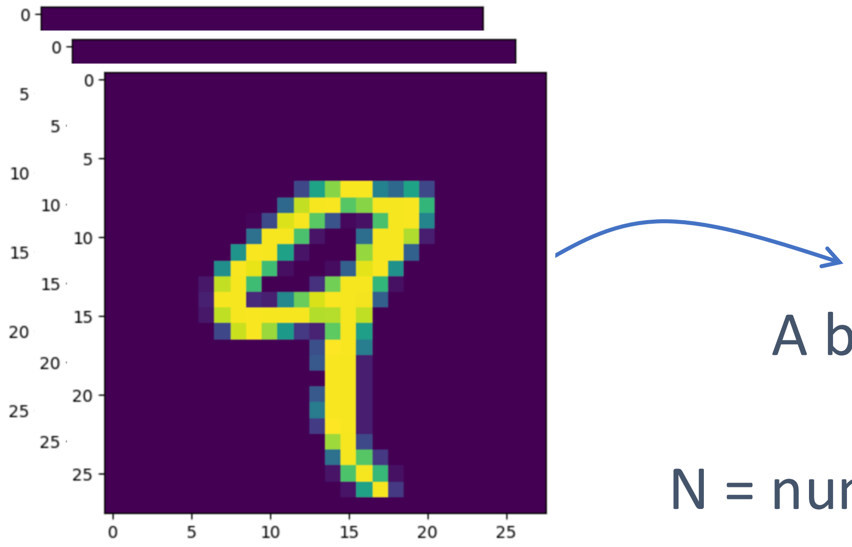
$$y = x.A^T + b$$

Practical 3



A typical MNIST sample (28x28 image)

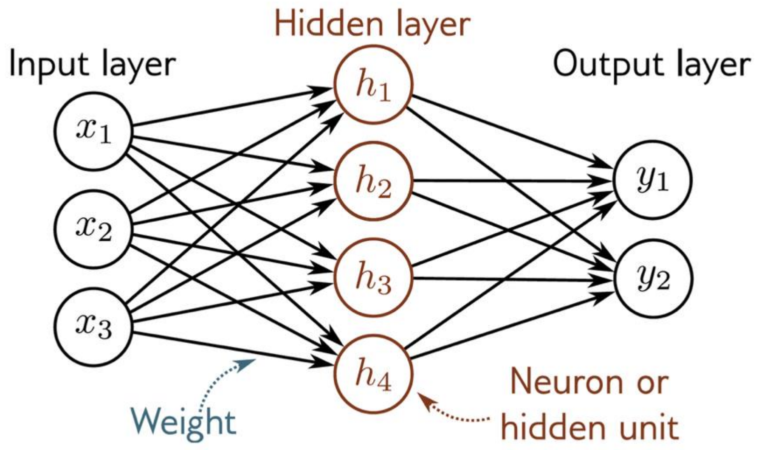
Practical 3



A batch is an N x 784 tensor

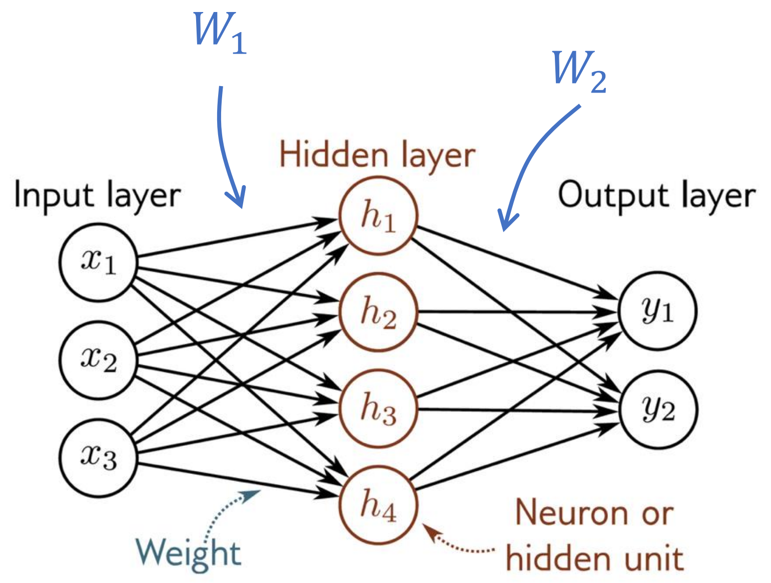
N = number of samples in the batch

Practical 3

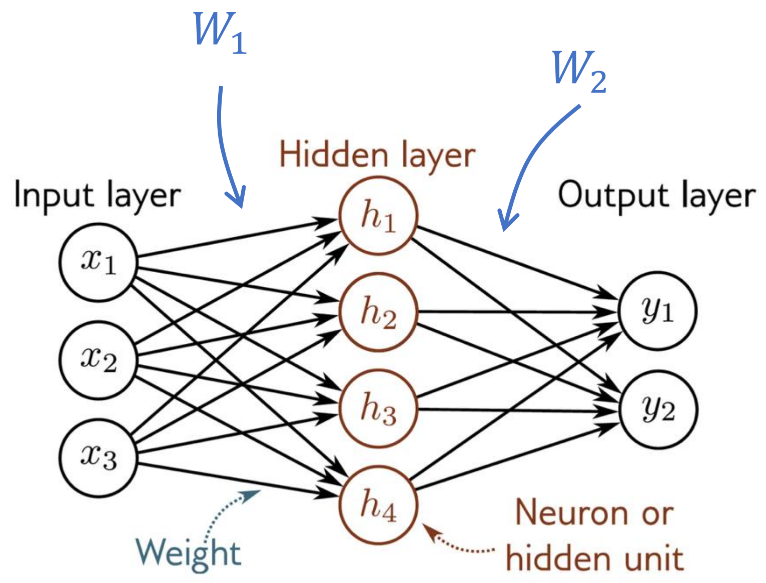


How many weight matrices are in a single layer MLP?

Practical 3


W_1
W_2
Hidden layer
Input layer
Output layer
x_1
x_2
x_3
h_1
h_2
h_3
h_4
y_1
y_2
Weight
Neuron or hidden unit

Practical 3

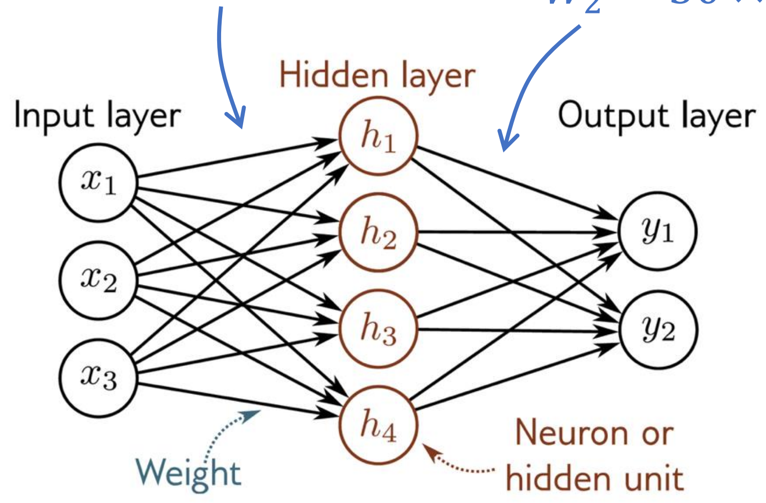


What are the matrix sizes with a hidden layer of dimension 50 and 10 output classes?

Practical 3

$W_1 = 784 \times 50$

$W_2 = 50 \times 10$

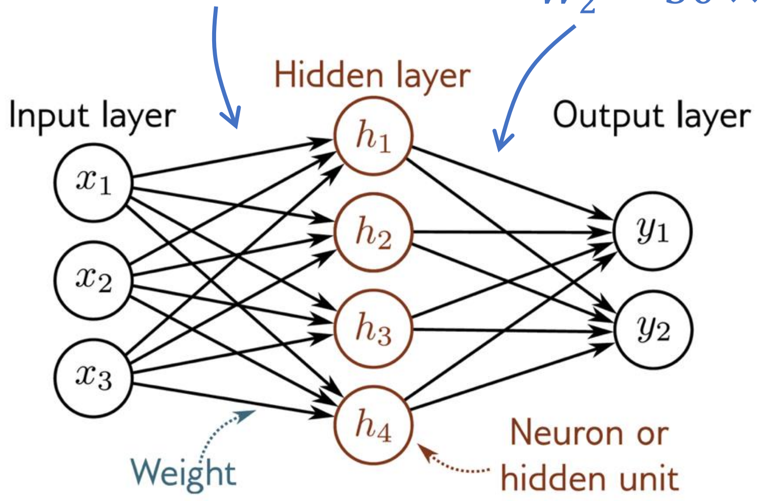


What are the matrix sizes with a hidden layer of dimension 50 and 10 output classes?

Practical 3

$W_1 = 784 \times 50$

$W_2 = 50 \times 10$



```
w1 = torch.randn((784, 50))
b1 = torch.randn((50))
w2 = torch.randn((50, 10))
b2 = torch.randn((10))
```

How do you implement the forward pass with a tanh activation?

Batch of 5 samples

```
x1 = train_input[:5]
y1 = train_target[:5]
z1 = x1 @ w1 + b1
h1 = sigma(z1)
z2 = h1 @ w2 + b2
h2 = sigma(z2)
l = loss(h2, y1)
```

w1 = 784 x 50

x1 = N x 784

z1 = N x 50

b1 = 1 x 50

Mission 1: implement sigma and loss (taking as input matrices)

Practical 3

How do you implement the backward pass?

```
x1 = train_input[:5]

y1 = train_target[:5]

z1 = x1 @ w1 + b1

h1 = sigma(z1)

z2 = h1 @ w2 + b2

h2 = sigma(z2)

l = loss(h2, y1)
```

```
dl = 1.0
```

Practical 3

How do you implement the backward pass?

```
x1 = train_input[:5]
y1 = train_target[:5]
z1 = x1 @ w1 + b1
h1 = sigma(z1)
z2 = h1 @ w2 + b2
h2 = sigma(z2)
l = loss(h2, y1)
```

```
dh2 = dloss(h2, y1) * dl
```

Practical 3

How do you implement the backward pass?

```
x1 = train_input[:5]

y1 = train_target[:5]

z1 = x1 @ w1 + b1

h1 = sigma(z1)

z2 = h1 @ w2 + b2

h2 = sigma(z2)

l = loss(h2, y1)
```

```
dz2 = dsigma(z2) * dh2
```

Practical 3

How do you implement the backward pass?

```
dl = 1.0

dh2 = dloss(h2, y1) * dl

dz2 = dsigma(z2) * dh2

…

dw1 = …

db1 = …
```

Mission 2: implement dsigma and dloss (taking as input matrices)

Practical 3

How do you derive the linear operation?

dw1 = ??? given dz1

z1 = x1 @ w1 + b1

Practical 3

How do you derive the linear operation?

$$\begin{tikzcd} \texttt{z1 = x1 @ w1 + b1} & \texttt{dw1 = ??? given dz1} \arrow[l, bend left] \end{tikzcd}$$

If all these variables were 1D real numbers:

```
z1 = x1 * w1 + b1

dw1 = x1 * dz1
```

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

Practical 3

How do you derive the linear operation?

```
z1 = x1 @ w1 + b1   ←   dw1 = ??? given dz1
```

But these variables are matrices!

```
dw1 = x1.T @ dz1
```

(Not) understanding derivative of a matrix-matrix product.

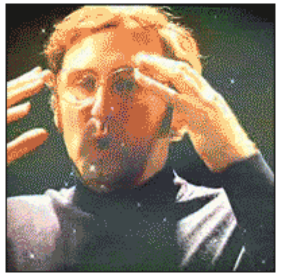
Practical 3

How do you derive the bias term?

$$\text{z1 = x1 @ w1 + b1} \xleftarrow{} \text{db1 = ??? given dz1}$$

Remember these variables are matrices!

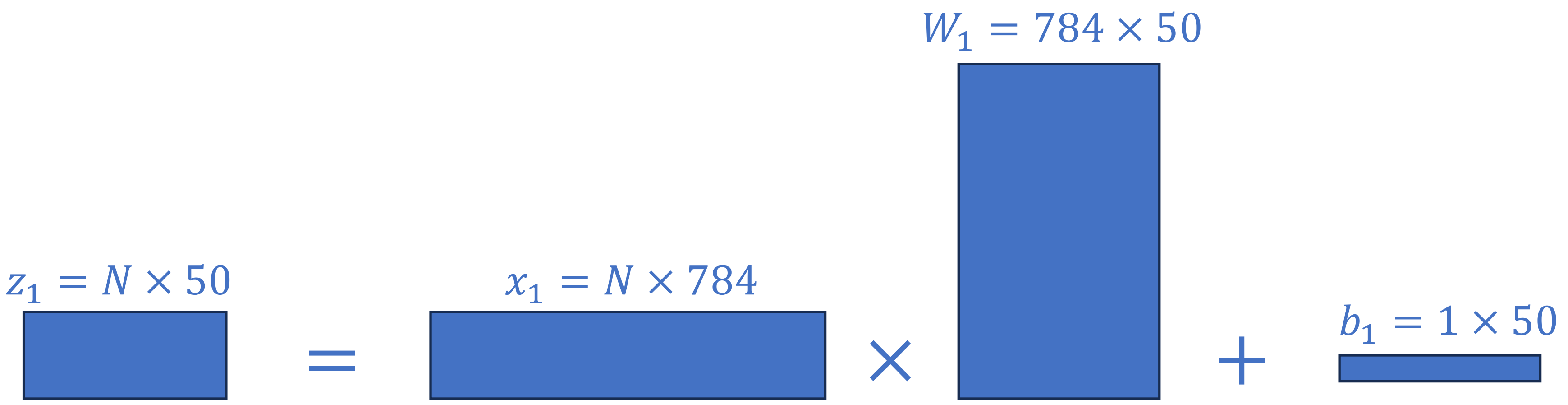
db1 = dz1 x 1...?



db1 is 1 x 50 and dz1 is N x 50

Practical 3

z1 = x1 @ w1 + b1

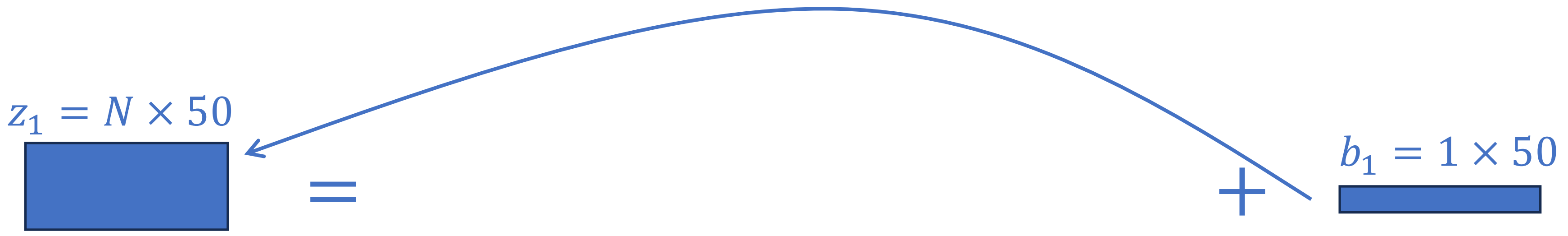


N = number of samples in the batch

Practical 3

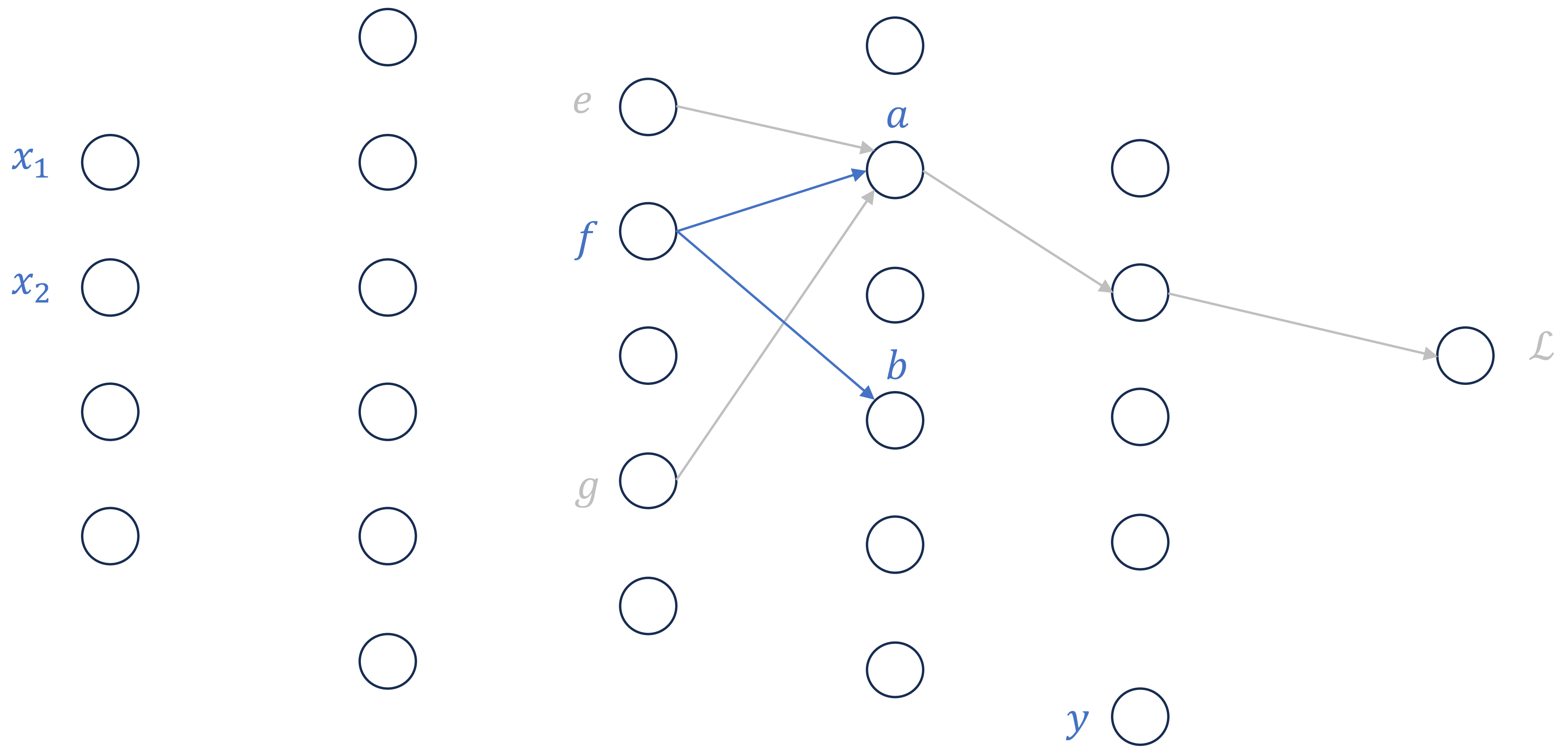
z1 = x1 @ w1 + b1

Broadcasting!
Bias is added to each row of the z1 matrix



The gradient flow

$\frac{\partial \mathcal{L}}{\partial a}$ and $\frac{\partial \mathcal{L}}{\partial b}$ contribute to $\frac{\partial \mathcal{L}}{\partial f}$



Practical 3

```
z1 = x1 @ w1 + b1

db1 = dz1.sum(axis=0, keepdim=True)
```

Practical 3

How do you sanity-check your implementation?

```
dl = 1.0
dh2 = dloss(h2, y1) * dl
cmp('h2',dh2,h2)
```

A utility that compares your gradient (dh2) to the actual gradient of h2

How do you sanity-check your implementation?

Wait, but how do we know the ground-truth for the gradient?

```
dl = 1.0

dh2 = dloss(h2, y1) * dl

cmp('h2',dh2,h2)
```

This is a pytorch tensor!

You can ask pytorch to maintain its gradient.

Practical 3

```
w1 = torch.randn((784, 50))
b1 = torch.randn((50,))
w2 = torch.randn((50, 10))
b2 = torch.randn((10,))
parameters = [w1, b1, w2, b2]
for p in parameters:
  p.requires_grad = True
  p.grad = None
```

$\leftarrow$ First, parameters should have a grad.

Practical 3

```
others = [h2,z2,h1,z1]
for t in others:
  t.retain_grad()
l.backward()
```

Ask pytorch to maintain grads for intermediate nodes

Magic! Call once, and it will populate all nodes with their gradient.

Practical 3

How do you sanity-check your implementation?

```
dl = 1.0
dh2 = ...
cmp('h2',dh2,h2)
dz2 = ...
cmp('z2',dz2, z2)
dw2 = ...
cmp('w2',dw2, w2)
db2 = ...
cmp('b2',db2, b2)
dh1 = ...
cmp('h1',dh1, h1)
dz1 = ...
cmp('z1', dz1, z1)
dw1 = ...
cmp('w1', dw1, w1)
db1 = ...
cmp('b1', db1, b1)
```

Practical 3

How do you implement gradient descent?

```
lr = 0.1
with torch.no_grad():
  w1 += -lr * dw1
  b1 += -lr * db1.squeeze()
  w2 += -lr * dw2
  b2 += -lr * db2.squeeze()
```

Mission 3: implement the gradient update

Practical 3

I’ll just call `.backward()`…

Backprop ninja

Practical 3

You doing backprop on softmax

(zero to hero by Karpathy)

Practical 3

You just unlocked `nn.Linear()` and `.backward()` on tensors!

Step 2: replace your loss with pytorch's backward

unchanged

New!

```
z1, h1, z2, h2 = forward(w1, b1, w2, b2, xb)
xloss = F.MSELoss()
lsi = xloss(h2, yb) * yb.nelement()

lsi.backward()
```

Practical 3

Step 3: finally, go full pytorch

```
class MLP(nn.Module):

  def __init__(self):
    super().__init__()
    self.layers = nn.ModuleList((???))

  def __call__(self, x):
    ???

  def __parameters__(self):
    return [p for layer in self.layers for p in layer.parameters]


model = MLP()
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
loss_fn = nn.MSELoss()
```