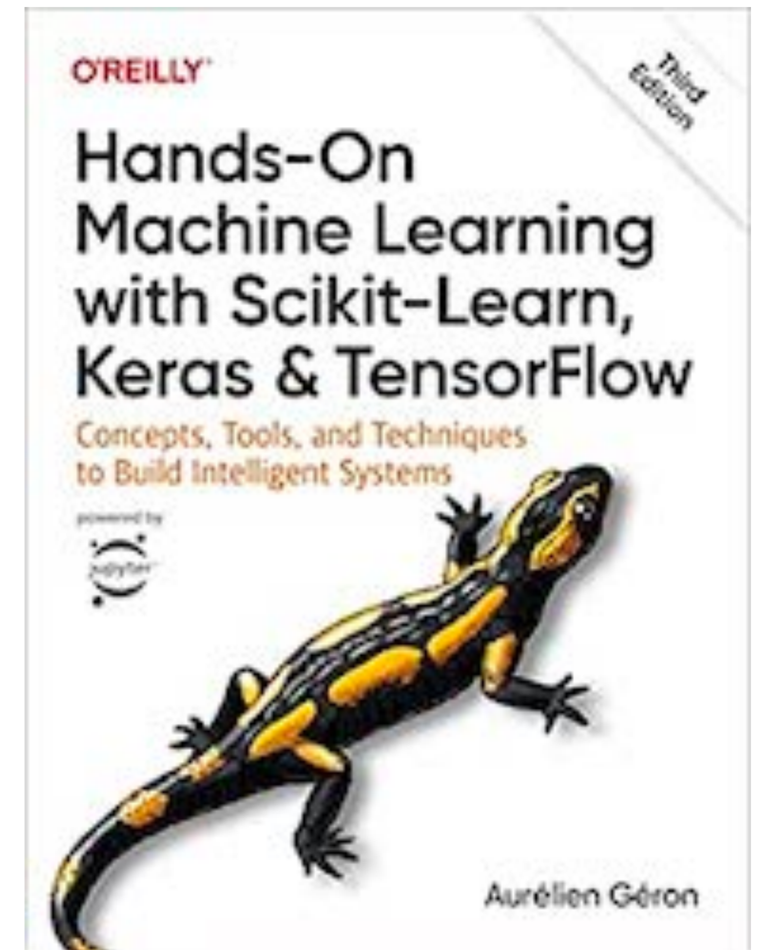
Machine Learning Security

11 Training Deep Neural Networks



Made Oct 27, 2023

Complex Problems

- In Ch 10, we made simple Artificial Neural Networks (ANNs) with just a few layers
- But more complex problems
  - Such as detecting hundreds of types of objects in high-resolution images
  - May need a deep ANN
- 10 or more layers
- Each with hundreds of neurons
- Hundreds of thousands of connections

Training Problems

- **You may be faced with the problem of gradients growing ever smaller or larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.**
- **You might not have enough training data for such a large network, or it might be too costly to label.**
- **Training may be extremely slow.**
- **A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances or if they are too noisy.**

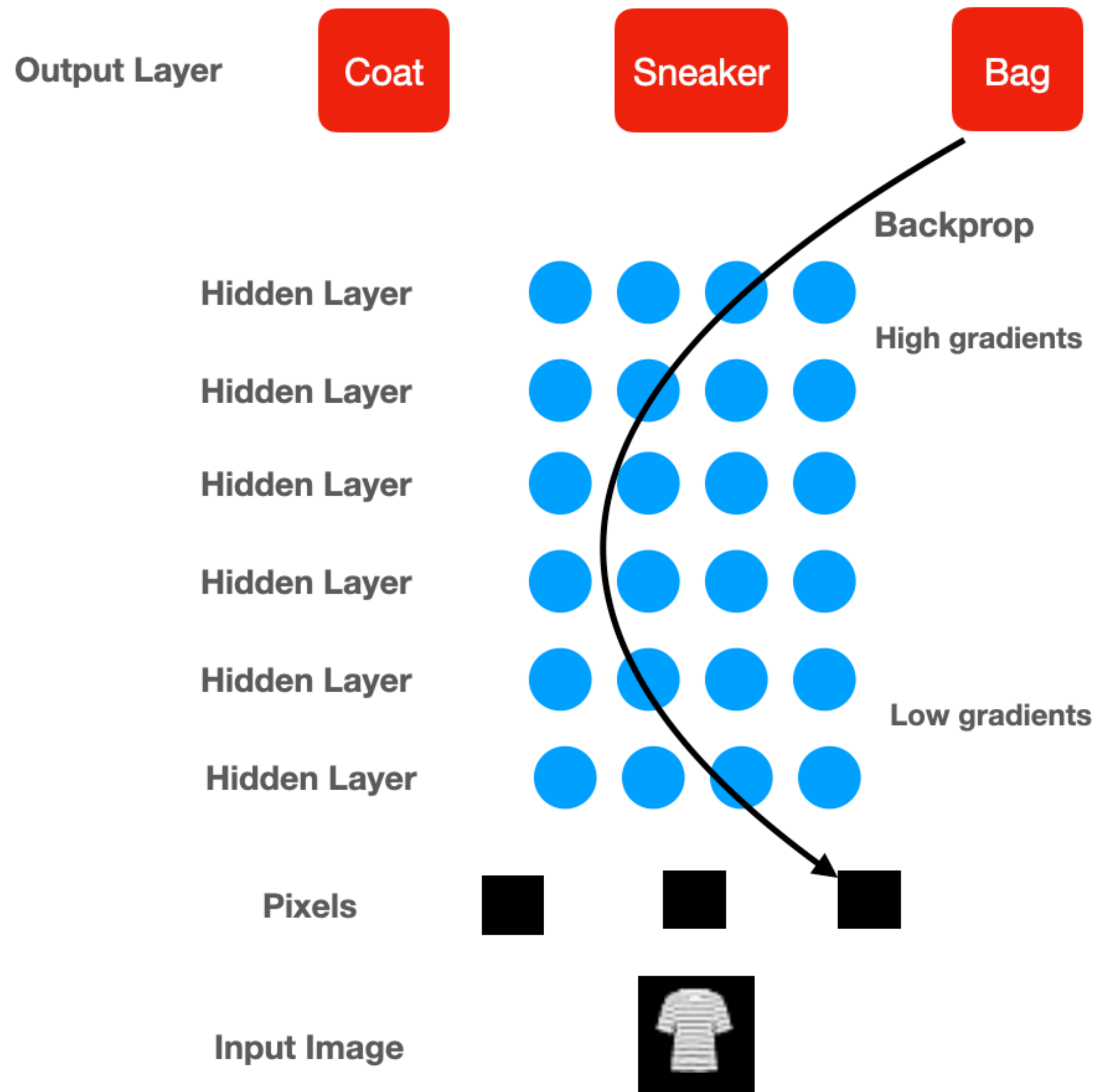
Topics

- The Vanishing/Exploding Gradients Problem
- Reusing Pretrained Layers
- Faster Optimizers
- Learning Rate Scheduling
- Avoiding Overfitting Through Regularization
- Summary and Practical Guidelines

The Vanishing/Exploding Gradients Problem

Vanishing Gradients

- The top layers are more important during backprop
- The lower layers have "vanishing gradients"
- So each cycle of learning doesn't change the lower layers much

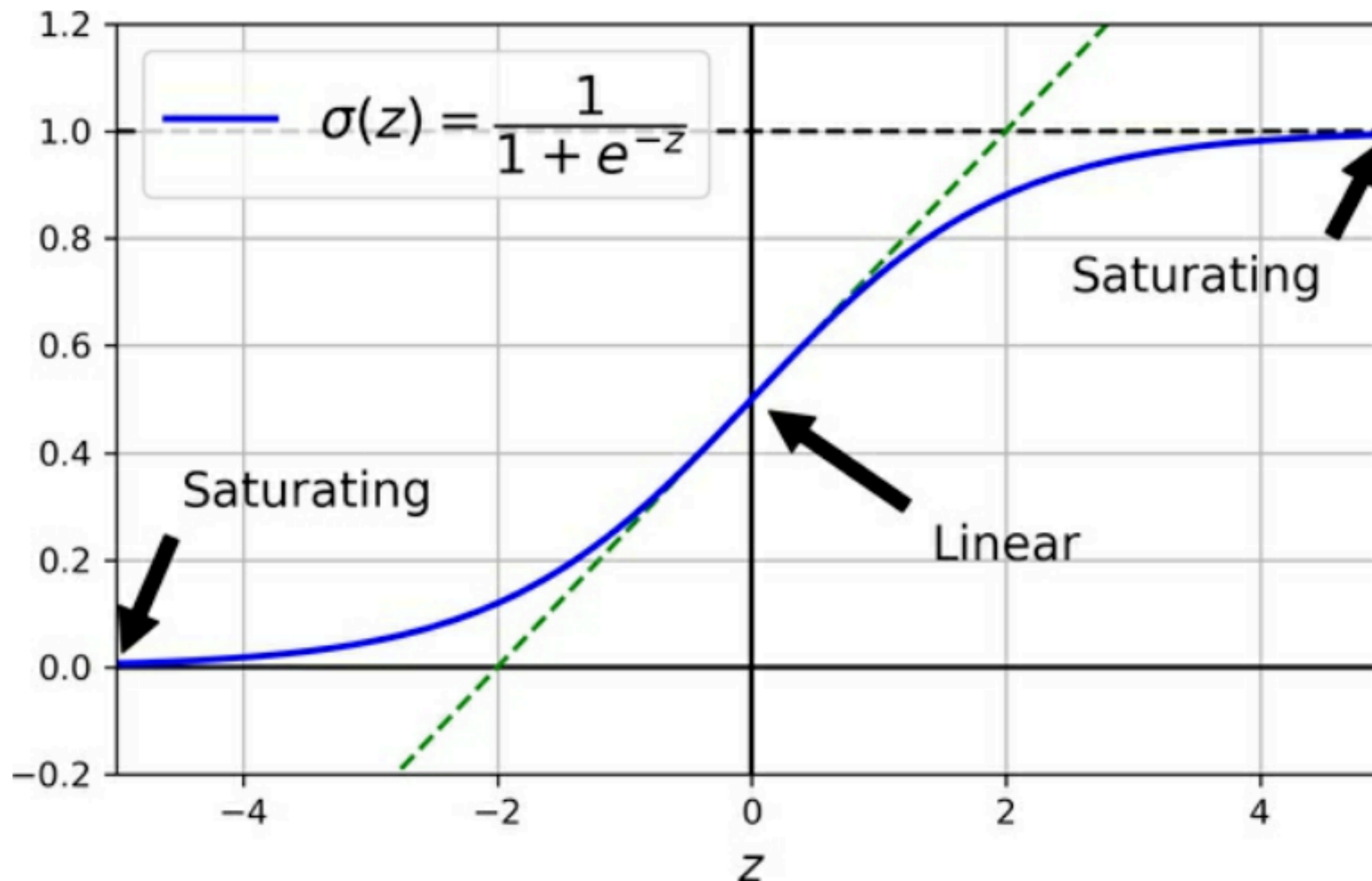


Exploding Gradients

- Sometimes the lower layers have larger "exploding gradients"
- Most often in recurrent neural networks
- The general problem is ***unstable gradients***
  - Different layers may learn at different speeds
- This problem was so serious, DNNs were mostly abandoned in the early 2000s
- In 2010, researchers explained the problem

Activation Function Saturation

- Large or small inputs enter ***saturation*** region
- Little or no gradient to guide learning



Glorot Initialization

- Connection weights of each layer must be initialized randomly from this distribution

Normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{\text{avg}}}$

Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{\frac{3}{fan_{\text{avg}}}}$

$$fan_{\text{avg}} = (fan_{\text{in}} + fan_{\text{out}}) / 2.$$

- ***fan-in*** is the number of inputs to a layer
- ***fan-out*** is the number of outputs from a layer

Initialization Parameters for Various Activation Functions

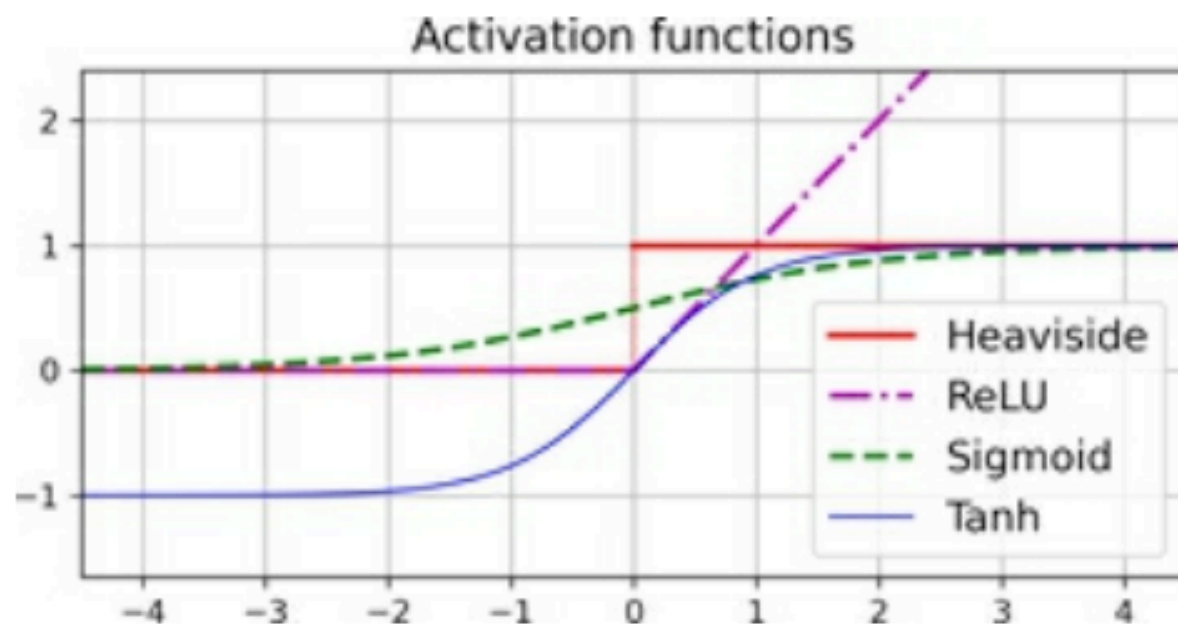
Table 11-1. Initialization parameters for each type of activation function

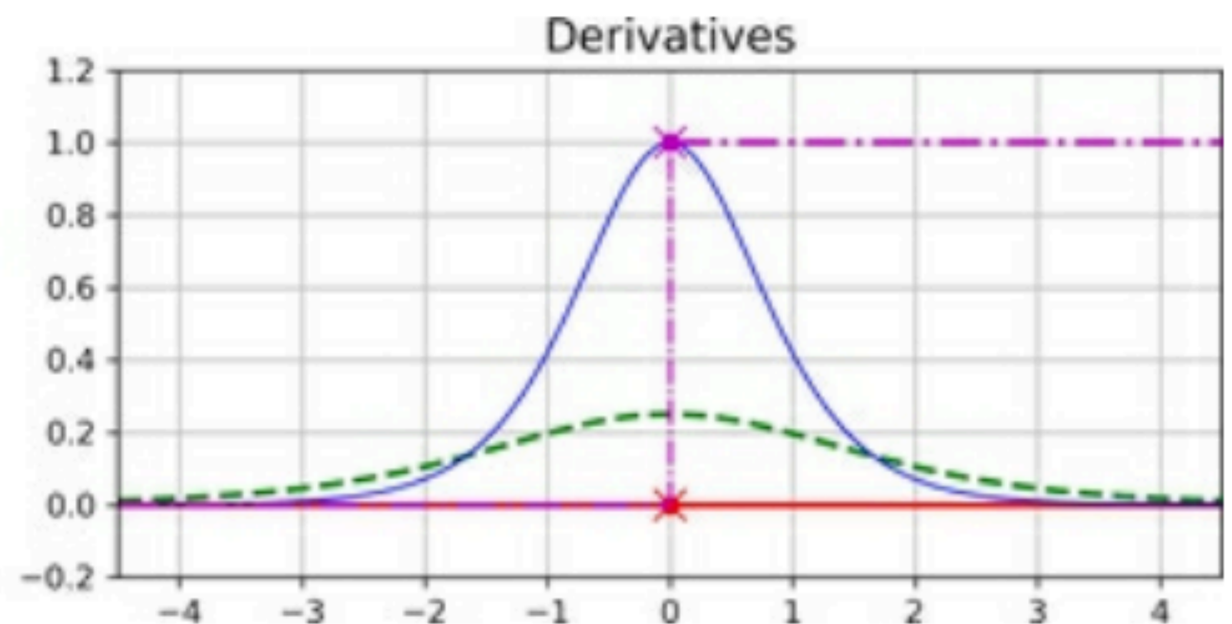
Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, sigmoid, softmax	$1 / fan_{avg}$
He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

- He initialization is best for the most powerful functions, like Swish

Dying ReLUs

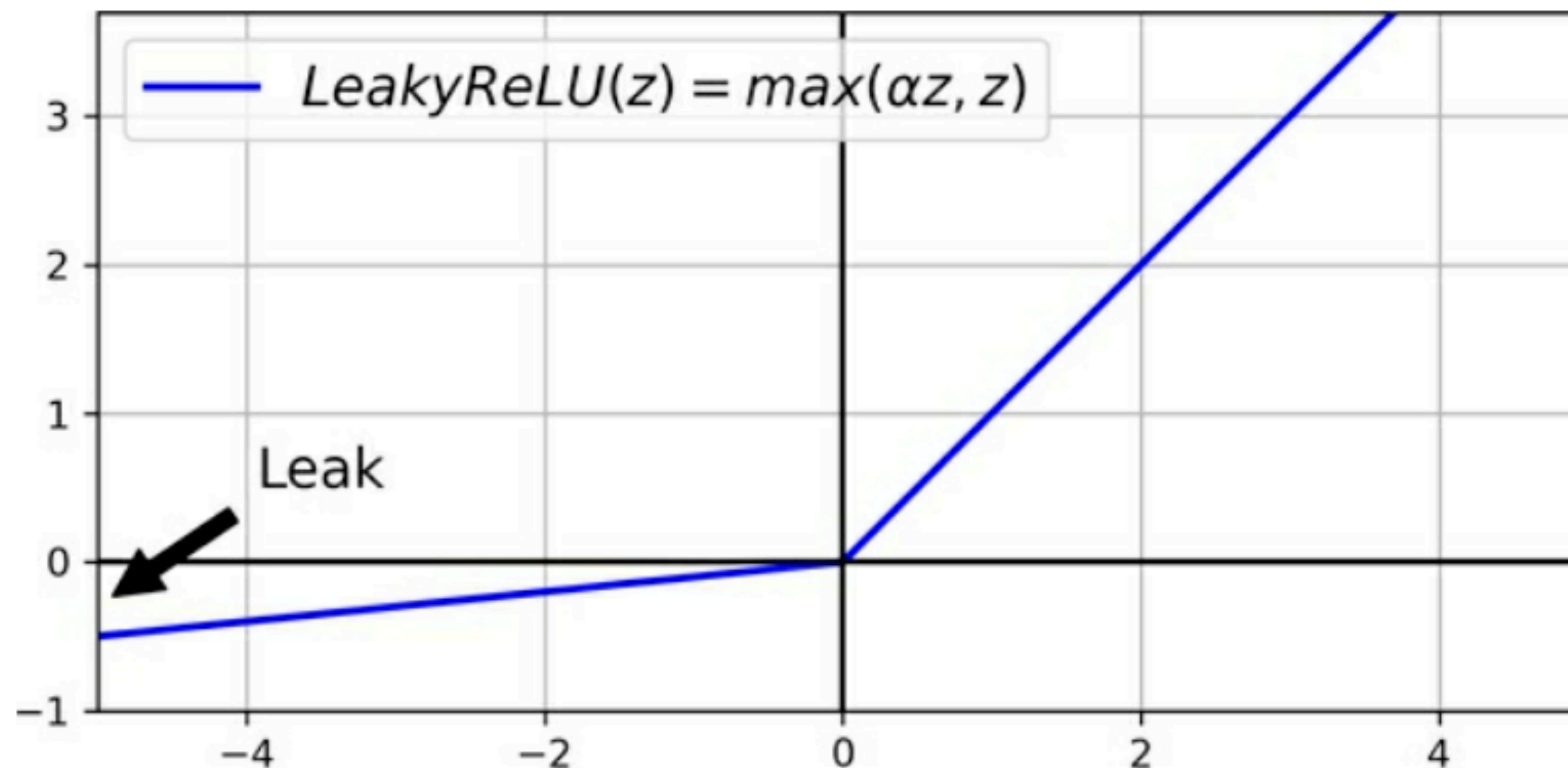
- ReLU has slope zero for input < 0
- During training, many neurons "die"
    - Their output is always zero





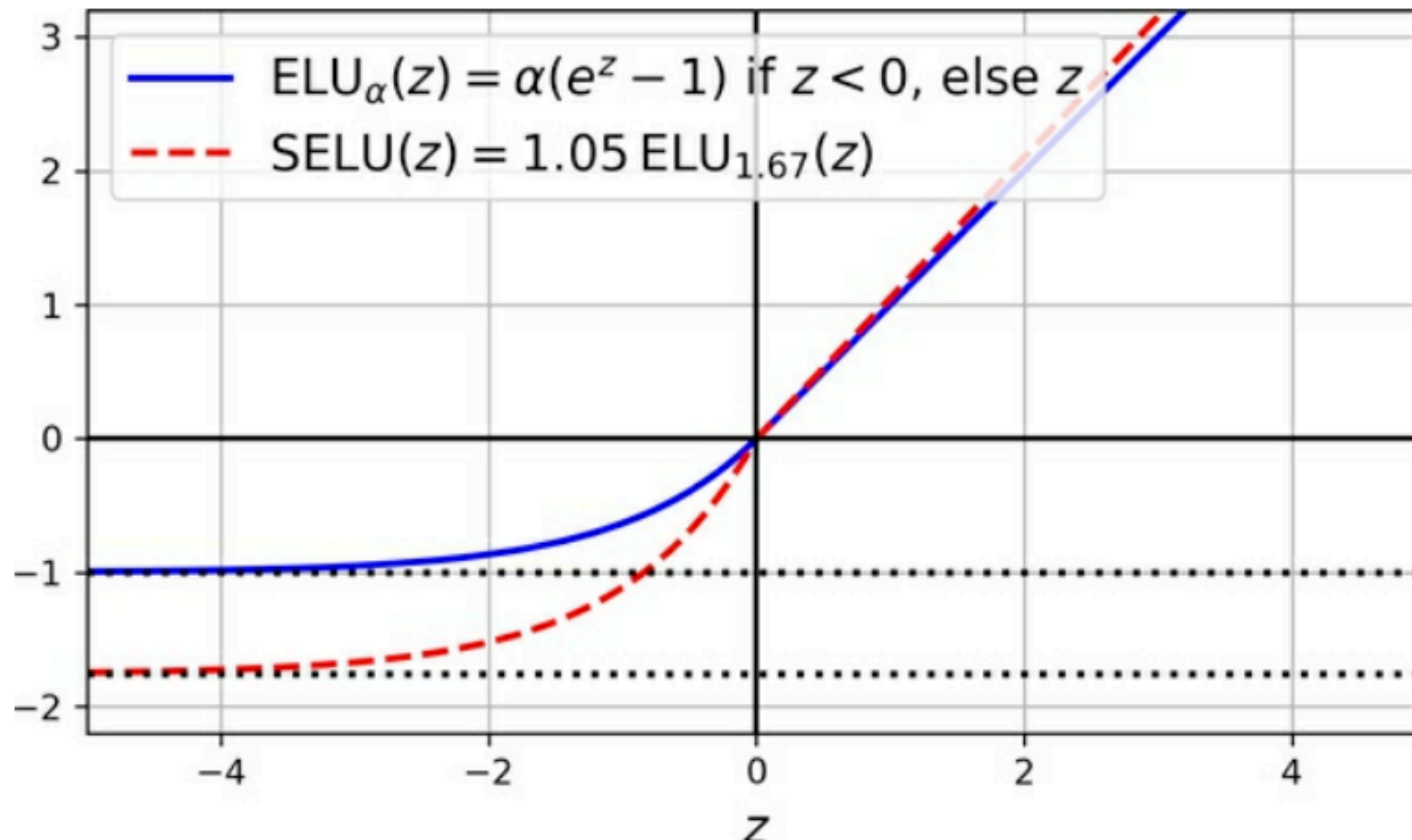
Leaky ReLU

- Doesn't have a region of zero slope
- Does have an abrupt change of slope at z=0
- This can make gradient descent bounce around



ELU and SELU

- Smooth variants of ReLU
- Exponential Linear Unit (ELU)
- Scaled ELU (SELU)



Self-Normalizing

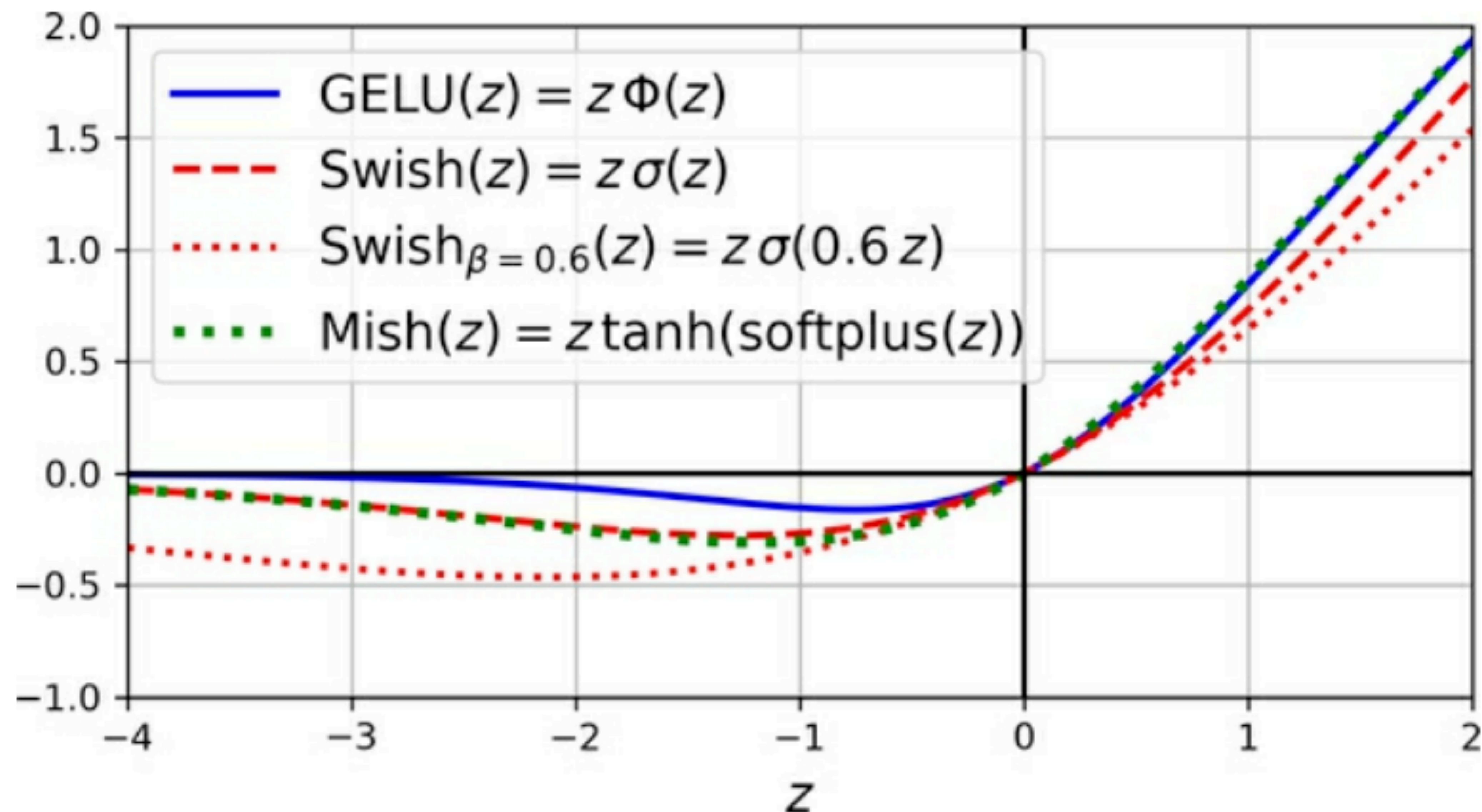
- A multi-layer perceptron using SELU for each hidden layer
  - will ***self-normalize***
    - Each layer's output will tend to a mean of 0 and std dev 1
    - Solving the vanishing/exploding gradients problem
  - BUT only when certain conditions apply (see next slide)
  - So SELU did not become popular

Conditions for Self-Normalization

- The input features must be standardized: mean 0 and standard deviation 1.
- Every hidden layer's weights must be initialized using LeCun normal initialization. In Keras, this means setting `kernel_initializer="lecun_normal"`.
- The self-normalizing property is only guaranteed with plain MLPs. If you try to use SELU in other architectures, like recurrent networks (see Chapter 15) or networks with *skip connections* (i.e., connections that skip layers, such as in Wide & Deep nets), it will probably not outperform ELU.
- You cannot use regularization techniques like ℓ_1 or ℓ_2 regularization, max-norm, batch-norm, or regular dropout (these are discussed later in this chapter).

GELU, Swish, and Mish

- ϕ is the Gaussian cumulative distribution function
- These work well for complex tasks



Batch Normalization

- He initialization and Swish reduces the danger of vanishing/ exploding gradients
  - But they may come back during training
- In 2015, a new technique was proposed: **batch normalization**
- Adds an operation before or after the activation function of each hidden layer
  - Zero-centers and normalizes each input
  - Scales and shifts the result with two new parameters per layer

Batch Normalization

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(300, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

- Doesn't matter for this small model
- Can greatly improve deeper networks

Batch Normalization

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 784)               3136
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 271,346
```

Batch Normalization

1. $$\boldsymbol{\mu}_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$$

2. $$\boldsymbol{\sigma}_B{}^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$$

3. $$\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B{}^2 + \varepsilon}}$$

4. $$\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

- **μ_B is the vector of input means, evaluated over the whole mini-batch *B* (it contains one mean per input).**
- ***m_B* is the number of instances in the mini-batch.**
- **σ_B is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).**
- **$\hat{\mathbf{x}}^{(i)}$ is the vector of zero-centered and normalized inputs for instance *i*.**
- **ε is a tiny number that avoids division by zero and ensures the gradients don't grow too large (typically 10^{-5}). This is called a *smoothing term*.**
- **γ is the output scale parameter vector for the layer (it contains one scale parameter per input).**
- **$\otimes$ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).**
- **β is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.**
- **$\mathbf{z}^{(i)}$ is the output of the BN operation. It is a rescaled and shifted version of the inputs.**

Predictions

- During training, batch normalization is recalculated for each batch
- Prediction for a single instance can't do that
  - No way to calculate mean or std dev for one instance
- The model calculates an exponential moving average of the mean and std dev during training
- And uses that after training is done

Batch Normalization Efficiency

- It makes models converge in up to 14 times fewer iterations
- But each iteration requires more calculations
- Often, after training, the weights of each layer can be modified to include the effect of the batch normalization layer

Hyperparameters

- **Momentum** smooths the exponential moving averages
- **Axis** controls how many means and std dev values are calculated for each batch

- Batch normalization has become one of the most-used layers in deep networks

Gradient Clipping

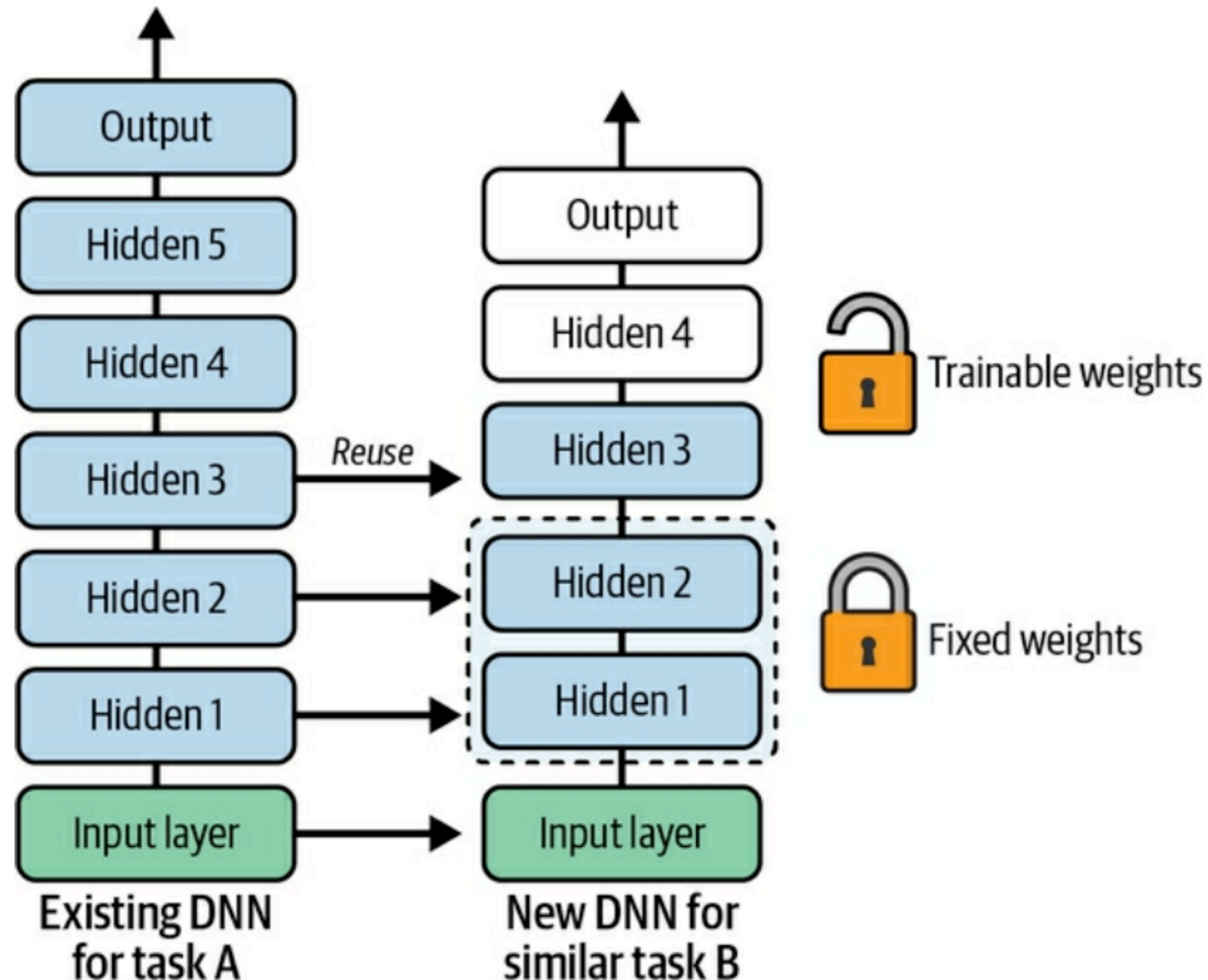
- Clip the gradients during backpropagation
  - So they never exceed a threshold
- Commonly used in recurrent neural networks
  - Where batch normalization is tricky

```
optimizer = tf.keras.optimizers.SGD(clipvalue=1.0)
model.compile([...], optimizer=optimizer)
```

Reusing Pretrained Layers

Transfer Learning

- Reuse lower layers from a model trained on a similar task
- Speeds up learning
- Requires less training data
- Output layer should usually be replaced to match the new task
- Inputs may require preprocessing to fit the size expected by the original model



Freezing Layers

- First try freezing all the reused layers
- Then unfreeze one or two of the top hidden layers
    - See if performance improves
- You may need to drop the top hidden layers
- If you have more training data, you can train more layers

When is Transfer Learning Best?

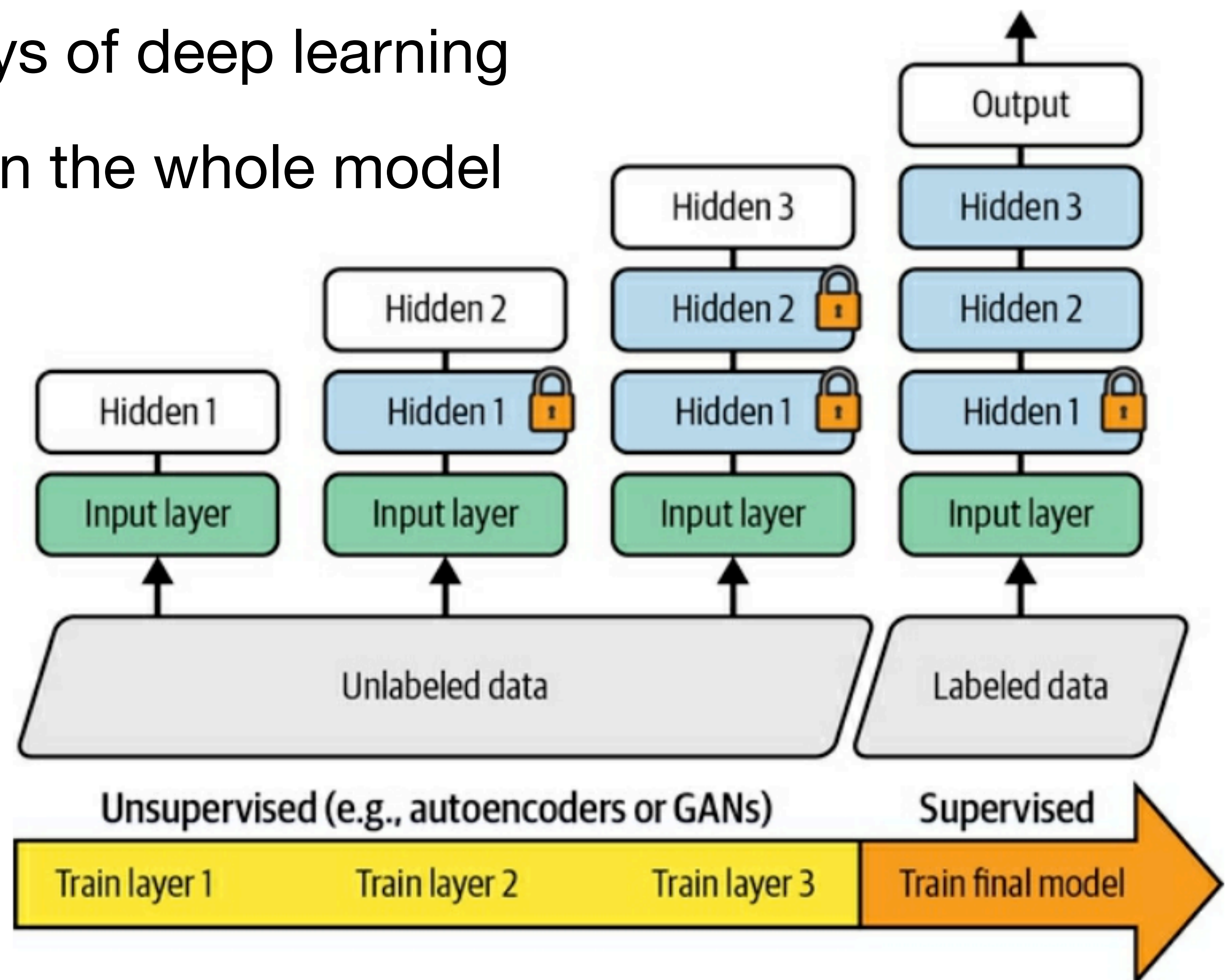
- Does not work well with small dense networks
  - Because they learn few patterns
  - Unlikely to be useful in other tasks
- Works best with deep comvolutional neural networks
  - Which tend to learn feature detectors that are more general

Unsupervised Pretraining

- Train an unsupervised model on unlabeled data, such as
  - **Autoencoder** which learns representations of the input data
  - Generative Adversarial Network (**GAN**) that generates data similar to the training data and learns to discriminate real from fake data
  - **Diffusion**, which learns to generate images from noise
- Reuse the lower layers of that model for your new task
- This technique succeeded in 2006 and led to the revival of neural networks and the success of deep learning

Greedy Layer-Wise Pretraining

- Used in the early days of deep learning
- Train one layer at a time
- Used in the early days of deep learning
- Now we can just train the whole model at once



Pretraining on an Auxiliary Task

- Consider face recognition
  - But you have only a few pictures of each individual
  - Pretrain on a lot of random pictures from the Web
  - Train on detecting whether two pictures are of the same person
  - The lower levels of that model will serve for your new one
- Consider Natural Language Processing
  - Start with a model trained on random Web data
  - Train higher levels on the new data you want

Microsoft 365 Copilot

Microsoft 365 Apps

Microsoft 365 Trust Boundary

Response + app commands

User prompt

Customer data is not stored or used to train the model

Large Language Model

Azure Open AI instance is maintained by Microsoft. OpenAI has no access to the data or the model.

Modified prompt

Azure OpenAI

RAI

LLM response

RAI is performed on input prompt and output results

Pre-processing

Grounding

Microsoft Graph

Semantic Index

Your context and content

emails, files, meetings, chats, calendars, and contacts

Grounding

Post-processing

Customer Microsoft 365 Tenant

Data flow (🔒 = all requests are encrypted via HTTPS)

1. User prompts from Microsoft 365 Apps are sent to Copilot
2. Copilot accesses Graph and Semantic Index for pre-processing
3. Copilot sends modified prompt to Large Language Model
4. Copilot receives LLM response
5. Copilot accesses Graph and Semantic Index for post-processing
6. Copilot sends the response, and app command back to Microsoft 365 Apps

- https://siliconangle.com/2023/10/18/companies-scrambling-keep-control-private-data-ai-models/

Self-Supervised Learning

- Automatically generate labels from the data itself
  - Such as by omitting a word from a sentence
  - Then training a model to guess the word

Kahoot!

Ch 11a

Faster Optimizers

Four Ways to Speed Up Training

- Good initialization for weights
- Good activation function
- Batch normalization
- Reusing parts of a pretrained network

- New way: a faster optimizer

Optimizers

- Stochastic Gradient Descent (SGD)
  - The old, slower way
- These are all faster:
  - **Momentum**
  - **Nesterov Accelerated Gradient**
  - **AdaGrad**
  - **RMSProp**
  - **Adam**

SGD

- Regular gradient descent takes
  - Large steps when the slope is steep
  - Small steps when the slope is gentle
  - Can converge very slowly down a gentle slope

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

Θ is the weights

η is the learning rate

***J*(Θ)** is the cost function

Momentum

- Imagine a bowling ball rolling down a hill
- ***Momentum optimization***
  - includes momentum from previous gradients
  - Gradient determines acceleration, not speed
- Escapes from plateaus much faster than SGD

$$
\begin{aligned}
&1. \qquad \mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
&2. \qquad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}
\end{aligned}
$$

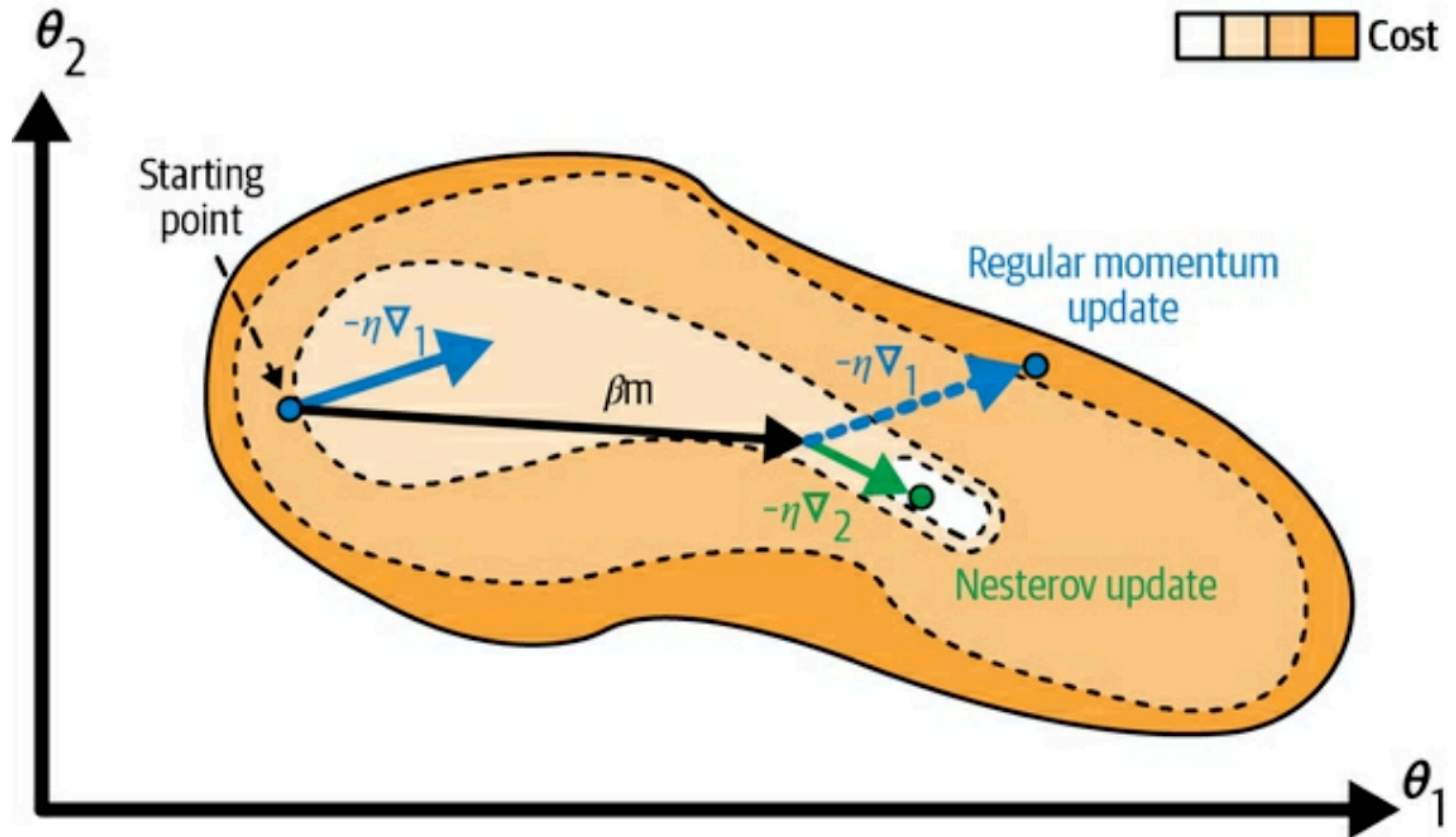
β is the **momentum** (friction, typically 0.9)

Nesterov Accelerated Gradient

- Calculates the gradient looking ahead in the direction of the momentum
- Almost always faster than regular momentum optimization

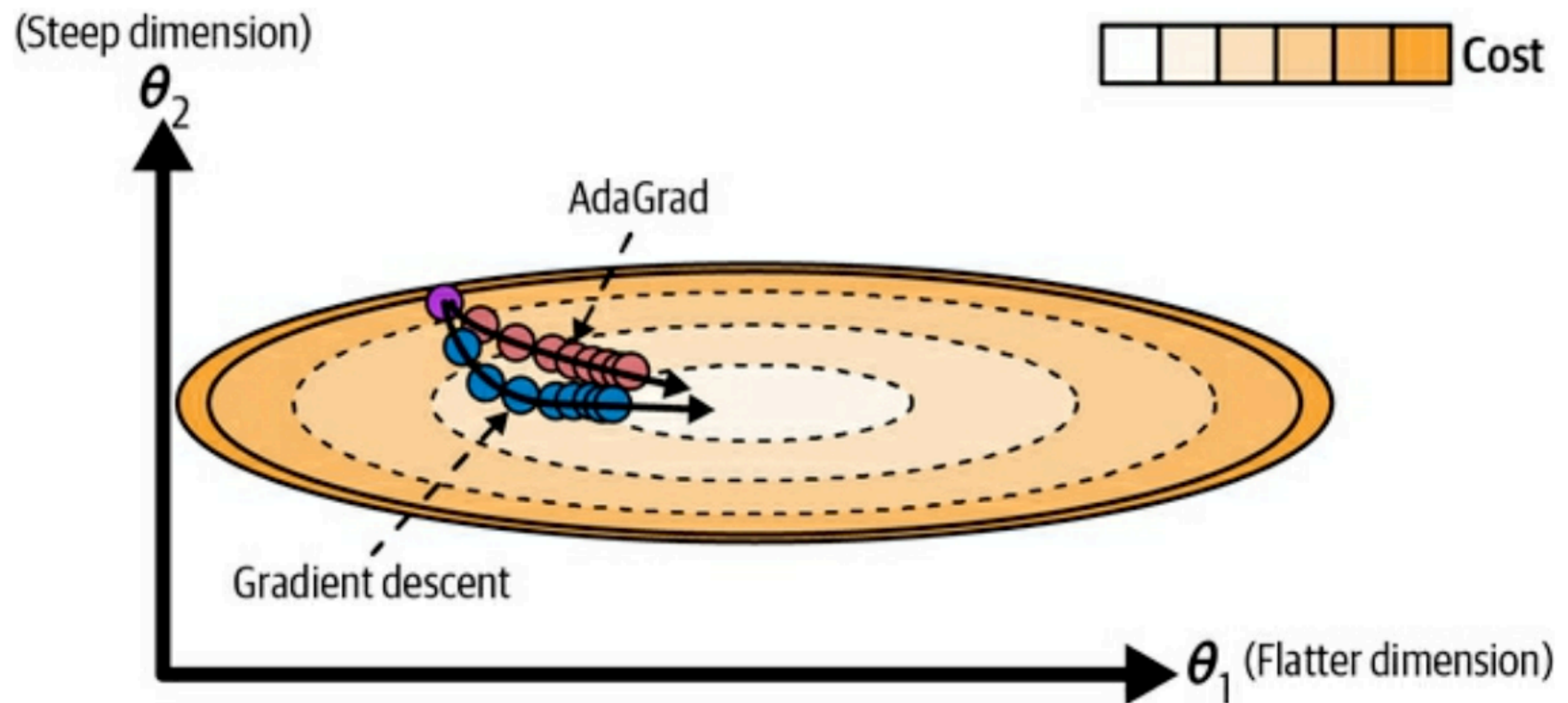
$$
\begin{aligned}
&1. \quad \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta \mathbf{m}) \\
&2. \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}
\end{aligned}
$$

Nesterov Accelerated Gradient



AdaGrad

- Moves more towards the global optimum
  - Instead of just down the steepest slope



AdaGrad

- Works for simple quadratic problems
- Often stops too early when training neural networks
- Scales the steps down and stops before reaching the optimum
- Don't use it

RMSProp

- Improves AdaGrad by only accumulating recent iterations in the gradient calculation
- Almost always works better than AdaGrad
- Was the preferred method until Adam was developed

Adam

- Adaptive Moment Estimation
- Combines features of momentum optimization and RMSProp
- Has hyperparameters

$\boldsymbol{\beta_1}$ is the **momentum** (typically 0.9)

$\boldsymbol{\beta_2}$ is the **decay rate** (typically 0.999)

$\boldsymbol{\varepsilon}$ is a smoothing term (typically 10^{-7})

$\boldsymbol{\eta}$ is the learning rate (default 0.001)

AdaMax

- Modifies the loss function of Adam
- In general, Adam performs better, so this isn't worth using most of the time

Nadam

- Adam optimization plus the Nesterov trick
- Often converges slightly faster than Adam

AdamW

- A variant of Adam
- Adds a regularization technique called ***weight decay***
- Multiplies the weights at each training iteration by a decay factor such as 0.99

***

is best

Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`AdaMax`	***	** or ***
`Nadam`	***	** or ***
`AdamW`	***	** or ***

Training Sparse Models

- Most parameters are zero
- Apply strong ℓ_1 regularization
  - As shown below, includes the sum of the absolute value of all the weights in the loss function

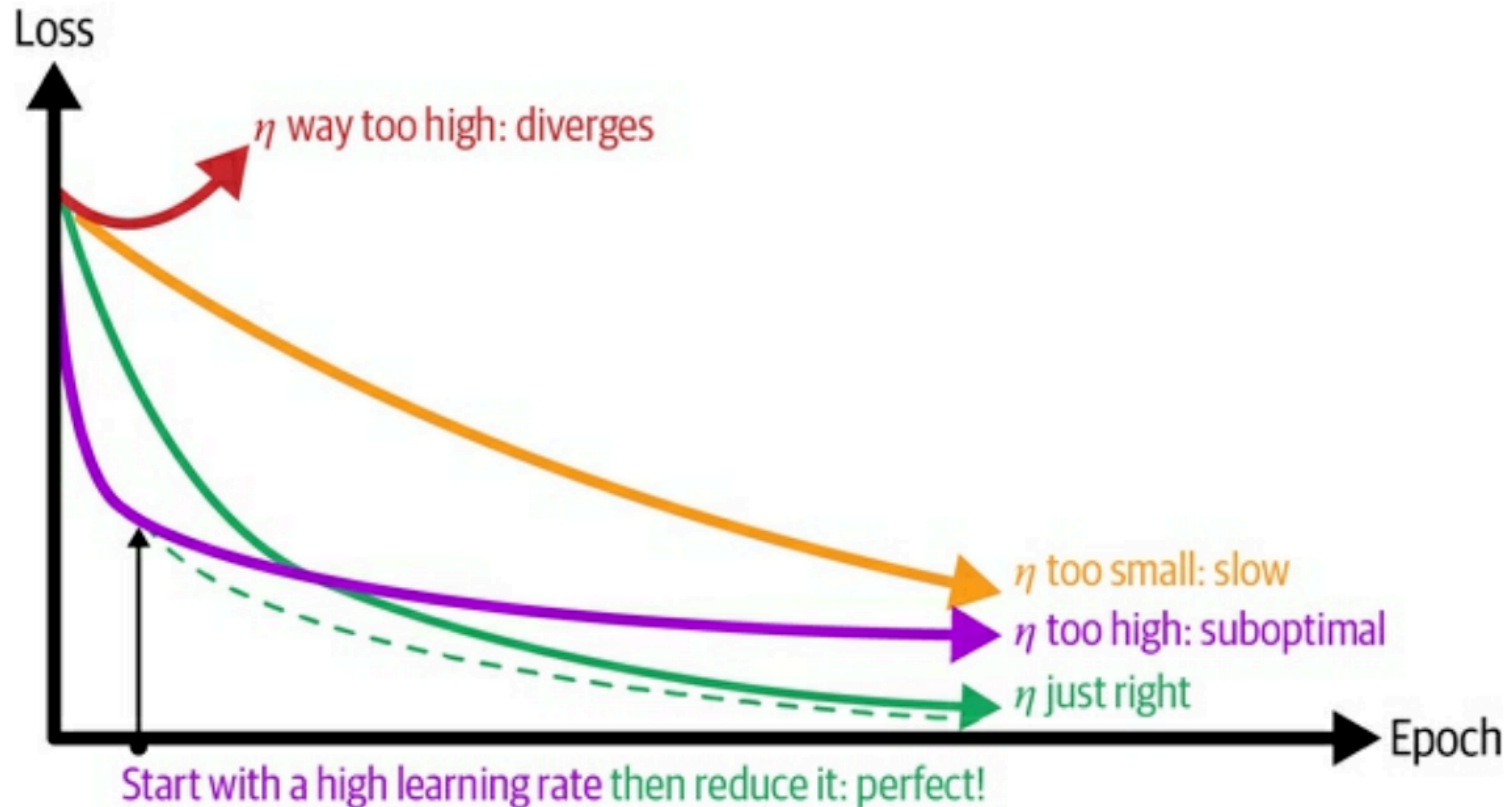
Equation 4-10. Lasso regression cost function

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + 2\alpha \sum_{i=1}^{n} |\theta_i|$$

Learning Rate Scheduling

Learning Rate

- Using a ***schedule*** to adjust learning rate is best



Schedules

- **Power scheduling**
  - Drops gradually after each step
- **Exponential scheduling**
  - Gradually drops by a factor of 10 every ***s*** steps
- **Piecewise constant scheduling**
  - Constant learning rate for a number of epochs, then a different rate for another number of epochs, etc.
- **Performance scheduling**
  - Measure the validation error, reduce the learning rate when it stops dropping

Schedules

- **1cycle scheduling**
  - Introduced in 2018
  - Increases learning rate by a factor of 10 during first half of training
  - Then decreases it for the second half of training
  - Converges much faster than other methods

Avoiding Overfitting Through Regularization

Regularization Techniques

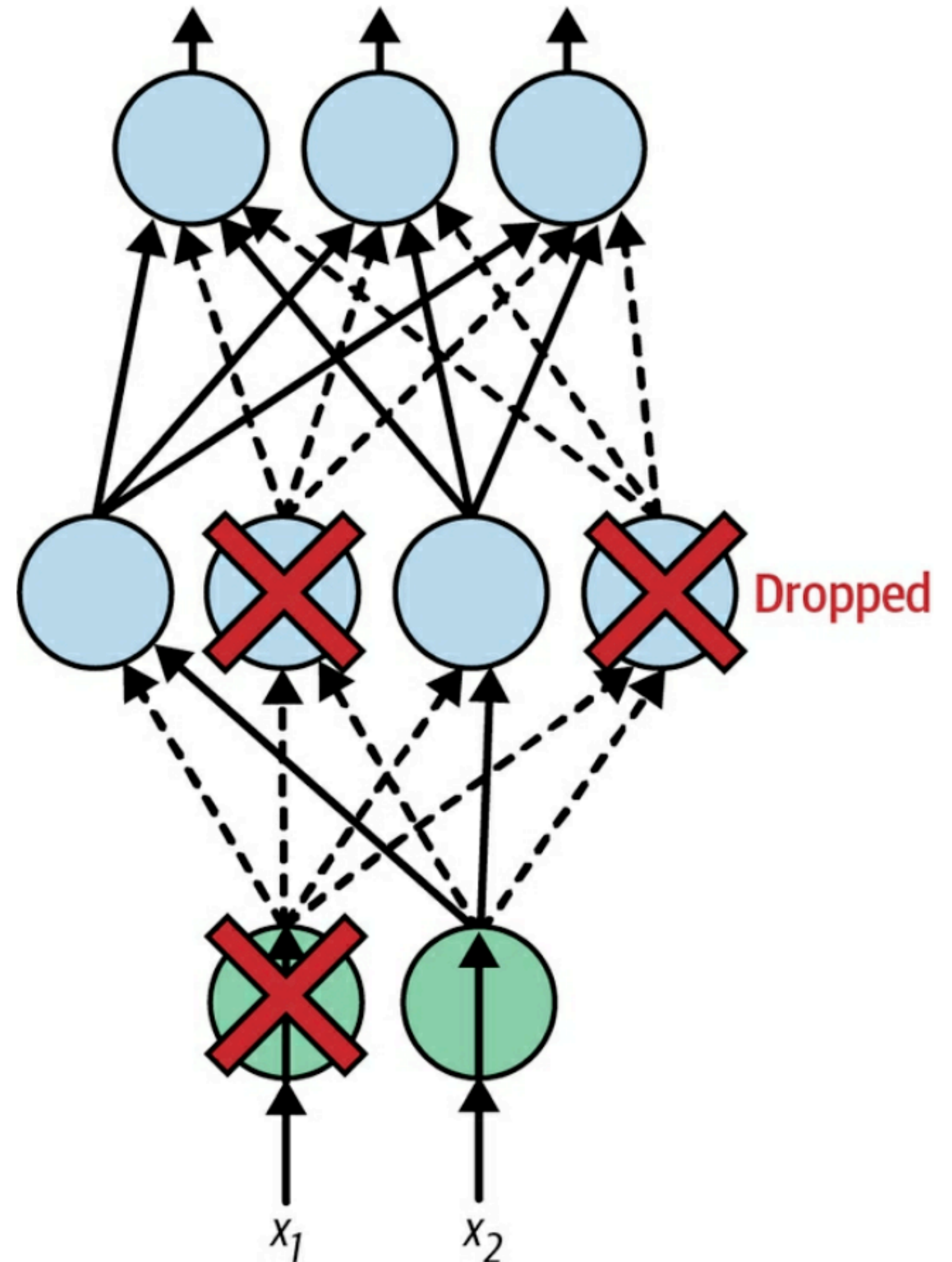
- Early stopping
- ℓ_1 and ℓ_2 regularization
  - ℓ_1 includes the sum of the absolute value of all the weights in the loss function
    - Ends up with a sparse model
  - ℓ_2 includes the sum of the squares of all the weights in the loss function
    - Good for SGD, momentum optimization, and Nesterov momentum optimization
    - Not good for Adam and its variants

Dropout Regularization

- A popular regularization technique for deep neural networks
- Provides 1%-2% accuracy boost
- At each training step, some neurons "drop out"
  - They are ignored for this step
- Probability of dropout is ***p*** (typically 10% - 50%)
- Produces a more robust network that generalizes better

Dropout Regularization

- In practice, only apply dropout to top one to three layers
- Excluding the output layer



Other Regularization Methods

- **Monte Carlo (MC) Dropout**
  - Averages many models with different random dropouts
- **Max-Norm Regularization**
  - Forces sum of all the squared weights to stay below a maximum

Summary and Practical Guidelines

Most Deep Neural Networks

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; batch norm if deep
Regularization	Early stopping; weight decay if needed
Optimizer	Nesterov accelerated gradients or AdamW
Learning rate schedule	Performance scheduling or 1cycle

A Simple Stack of Dense Layers

Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Nesterov accelerated gradients
Learning rate schedule	Performance scheduling or 1cycle

Kahoot!

Ch 11b