Hands-on Machine Learning

11. Training Deep Neural Networks

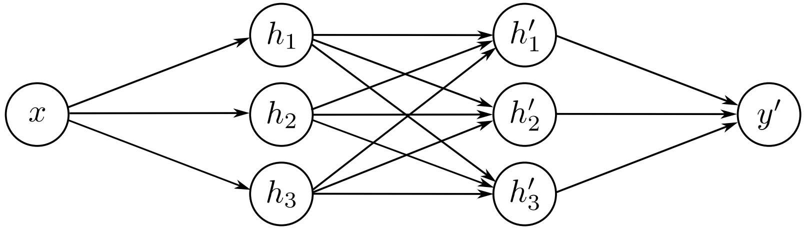
Challenges of Training a DNN

- Challenges of training a deep neural network:
  - You may face 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.

1. Vanishing/Exploding Gradients Problems

Vanishing Gradients Problem

- The backpropagation algorithm's backward pass works by going from the output layer to the input layer, propagating the error gradient along the way.



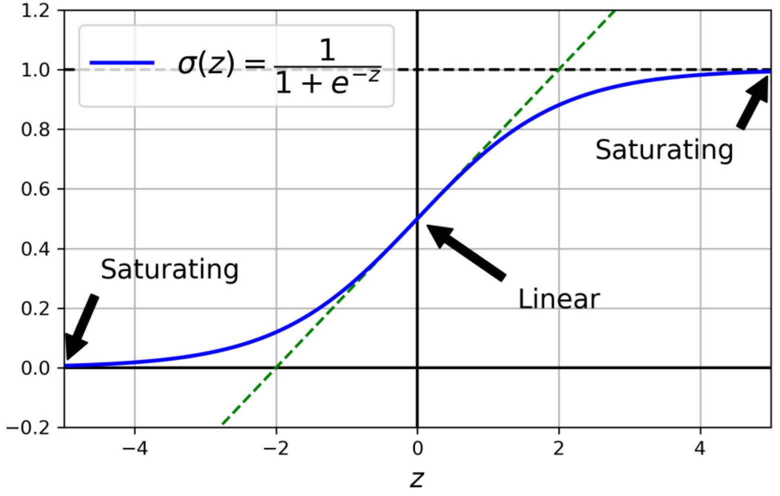
- Gradients often get smaller as the algorithm progresses down to the lower layers, and the gradient descent update leaves the lower layers' connection weights almost unchanged, and training never converges to a good solution.
  - This is called the *vanishing gradients* problem.

Exploding Gradients Problem

- In some cases, the gradients can grow bigger until layers get significantly large weight updates and the algorithm diverges.
  - This is the *exploding gradients* problem, which surfaces most often in recurrent neural networks (RNNs).
- Deep neural networks suffer from unstable gradients: different layers may learn at widely different speeds.
- Glorot and Bengio found a few suspects:
  - the popular sigmoid (logistic) activation function
  - the weight initialization technique that was most popular at the time, i.e. a standard normal distribution $N(0,1)$.

Sigmoid Saturation Problem

- When inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0 (i.e., the curve is flat at both extremes).
- When backpropagation starts it has almost no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.



Glorot Initialization

- We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients.
  - the signal shouldn’t die out, nor should it explode and saturate.
- The signal flows properly, if the variance of the outputs of each layer is equal to the variance of its inputs, and gradients have equal variance before and after flowing through a layer in the reverse direction.
- It is not possible to guarantee both unless the layer has an equal number of inputs and outputs (fan_{in} and fan_{out} of the layer), but Glorot and Bengio proposed a good compromise that has proven to work very well in practice.

Glorot Initialization

- The connection weights of each layer must be initialized randomly using
  - a normal distribution with mean 0 and variance $\sigma^2 = 1/fan_{avg}$
  - or a uniform distribution between $-r$ and r, with $r = \sqrt{3/fan_{avg}}$

  where $fan_{avg} = (fan_{in} + fan_{out})/2$.
- *LeCun initialization*: replace fan_{avg} with fan_{in}.
- Using Glorot initialization can speed up training considerably.
- Some papers have provided similar strategies for different activation functions.
  - These strategies differ only by the scale of the variance and whether they use fan_{avg} or fan_{in}.

He Initialization

- The initialization strategy proposed for the ReLU activation function and its variants is called *He initialization.*
- For SELU, use Yann LeCun's initialization method, preferably with a normal distribution.

Initialization	Activation Functions	σ^2 (normal)	r (uniform)
Glorot	None, tanh, sigmoid, softmax	$1/fan_{avg}$	$\sqrt{3/fan_{avg}}$
He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2/fan_{in}$	$\sqrt{6/fan_{in}}$
LeCun	SELU	$1/fan_{in}$	$\sqrt{3/fan_{in}}$

Glorot Initialization

- By default, Keras uses Glorot initialization with a uniform distribution.
- When you create a layer, you can switch to He initialization by setting `kernel_initializer="he_uniform"` or `"he_normal"` like this:

```
dense = tf.keras.layers.Dense(50, activation="relu", kernel_initializer="he_normal")
```

- Alternatively, you can obtain any of the mentioned initializations using the `VarianceScaling` initializer.
- For example, if you want He initialization with a uniform distribution and based on fan_{avg} (rather than fan_{in}), you can use:

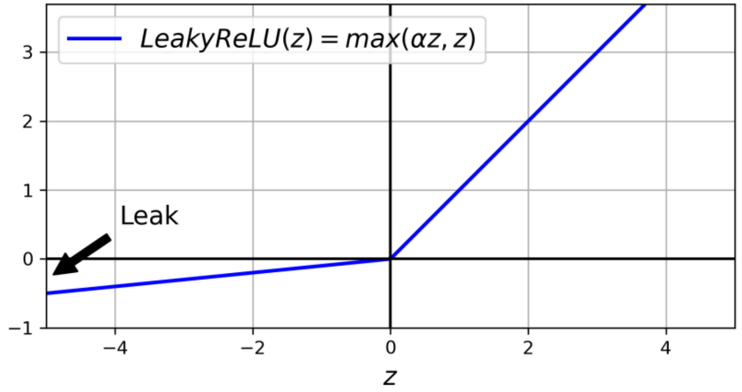
```
he_avg_init = tf.keras.initializers.VarianceScaling(scale=2., mode="fan_avg", distribution="uniform")
dense = tf.keras.layers.Dense(50, activation="sigmoid", kernel_initializer=he_avg_init)
```

Better Activation Functions

- Glorot and Bengio also showed that the problems with unstable gradients were in part due to a poor choice of activation function.
- In particular the showed ReLU is better than sigmoid because:
  - it does not saturate for positive values
  - it is very fast to compute
- *Dying ReLUs* problem: during training, some neurons effectively “die”, meaning they stop outputting anything other than 0.
  - you may find that half of your network’s neurons are dead, especially if you used a large learning rate.
- A neuron dies when its weights get tweaked such that the input of the ReLU function is negative for all instances in the training set.

Leaky ReLU

- The leaky ReLU activation function: $LeakyReLU_{\alpha}(z) = \max(\alpha z, z)$.
- The hyperparameter α defines how much the function "leaks": it is the slope of the function for $z < 0$. Having a slope for $z < 0$ ensures that leaky ReLUs never die.
- A 2015 paper by Bing Xu compared several variants of the ReLU activation function, and concluded that the leaky variants always outperformed the strict ReLU activation function.



Variants of Leaky ReLU

- *Randomized leaky ReLU* (RReLU): α is picked randomly in a given range during training and is fixed to an average value during testing.
  - RReLU performs fairly well and act as a regularizer, reducing the risk of overfitting the training set.

- *Parametric leaky ReLU* (PReLU): instead of being a hyperparameter, α becomes a parameter that can be modified by backpropagation like any other parameter.
  - PReLU strongly outperforms ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

Leaky ReLU in Keras

- Keras includes `LeakyReLU` and `PReLU` in the `tf.keras.layers` package.
- Like other ReLU variants, you should use *He initialization* with these.

```
leaky_relu = tf.keras.layers.LeakyReLU(alpha=0.2)  # defaults to alpha=0.3
dense = tf.keras.layers.Dense(50, activation=leaky_relu, kernel_initializer="he_normal")
```

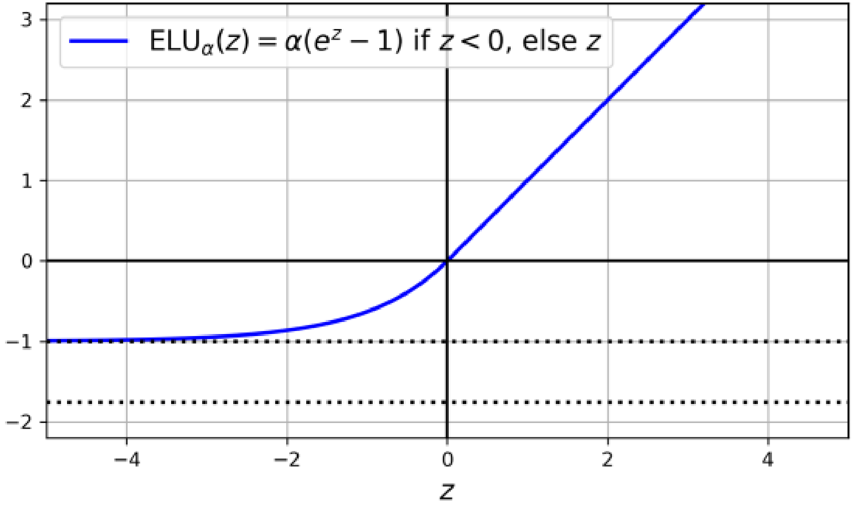
- You can use LeakyReLU as a separate layer in your model; it makes no difference for training and predictions:

```
model = tf.keras.models.Sequential([
    # [...]  # more layers
    tf.keras.layers.Dense(50, kernel_initializer="he_normal"),  # no activation
    tf.keras.layers.LeakyReLU(alpha=0.2),  # activation as a separate layer
    # [...]  # more layers
])
```

ELU

- In 2015, a paper by Clevert et al. proposed a new activation function, called the *Exponential Linear Unit* (ELU), that outperformed all the ReLU variants in the authors' experiments:
  - Training time was reduced, and the neural network performed better on the test set.

$$ELU_\alpha(z) = \begin{cases} \alpha(e^z - 1) & \text{if} \quad z < 0 \\ z & \text{if} \quad z \geq 0 \end{cases}$$

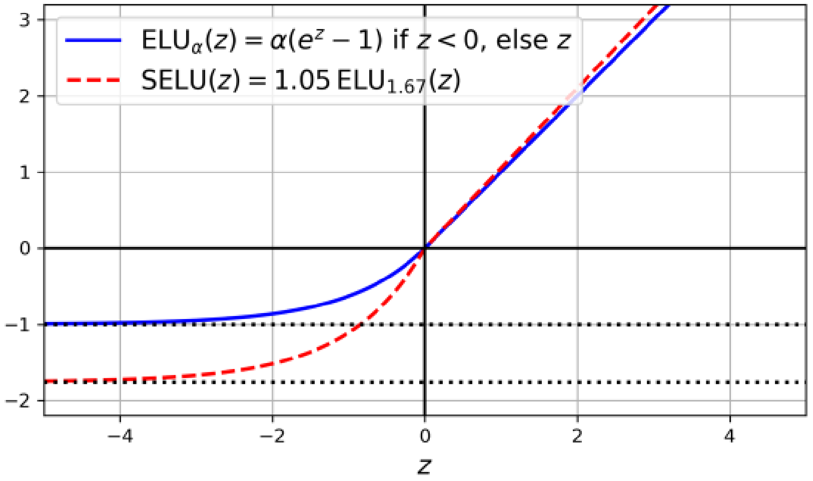


ELU

- The ELU function looks like the ReLU with a few major differences:
  - It takes on negative values when $z < 0$, which allows an average output closer to 0 and helps reduce the vanishing gradients problem.
  - The hyperparameter α is usually set to 1, but you can tweak it like any other hyperparameter.
  - ELU has a nonzero gradient for $z < 0$, which avoids the dead neurons problem.
  - If α is equal to 1 then the function is smooth everywhere, including around $z = 0$, which helps speed up gradient descent.
- The drawback of the ELU is that it is slower to compute than the ReLU and its variants (due to the exponential function).
  - faster convergence in training may compensate for slow computation, but at test time an ELU network will be slower than a ReLU network.

SELU

- A 2017 paper by Klambauer et al. introduced the *scaled ELU* (SELU).
- They showed that if you build a neural network composed exclusively of a stack of dense layers (i.e., an MLP), and if all hidden layers use the SELU activation function, then the network will *self-normalize*.
- *Self normalization*: the output of each layer will tend to preserve a mean of 0 and a standard dev of 1 during training, which solves the vanishing and exploding gradients problem.



Self-Normalization in SELU

- There are a few conditions for self-normalization to happen:
  - The input features must be standardized: $\mu = 0$, $\sigma = 1$.
  - Every hidden layer's weights must be initialized using LeCun normal initialization: `kernel_initializer="lecun_normal"`.
  - The self-normalizing property is only guaranteed with plain MLPs. If you try to use SELU in other architectures, like RNNs 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 significant constraints, so despite its promises, SELU did not gain a lot of traction.

GELU

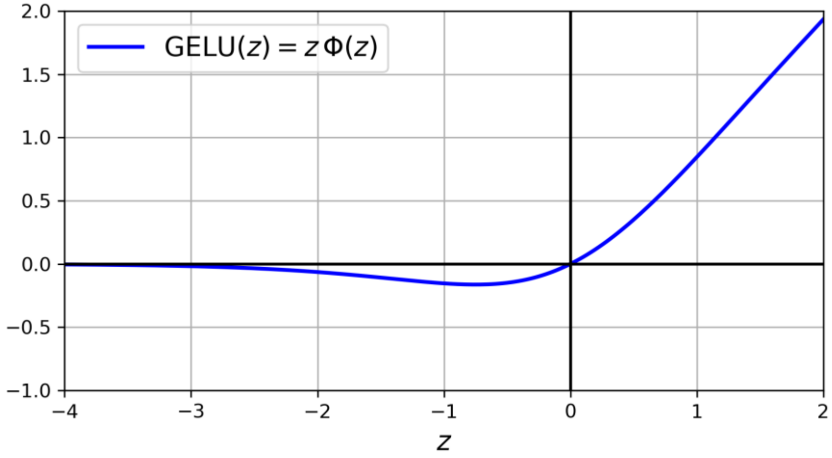
- *GELU* is another smooth variant of the ReLU activation function:

$$GELU(z) = z.\Phi(z)$$

  where Φ is the standard Gaussian cumulative distribution function.

- In practice, GELU often outperforms every other activation function discussed so far.

- It is computationally intensive, and the performance boost it provides is not always sufficient to justify the extra cost.

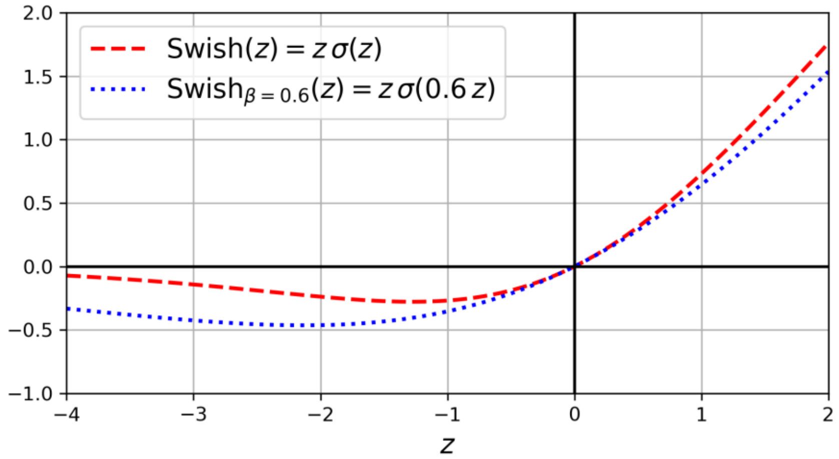


Swish

- Swish (a.k.a. the *sigmoid linear unit* or *SiLU*) is equal to $z.\sigma(z)$.
- It was later generalized to $Swish_\beta(z) = z.\sigma(\beta z)$ by adding an extra hyperparameter β to scale the sigmoid function's input.

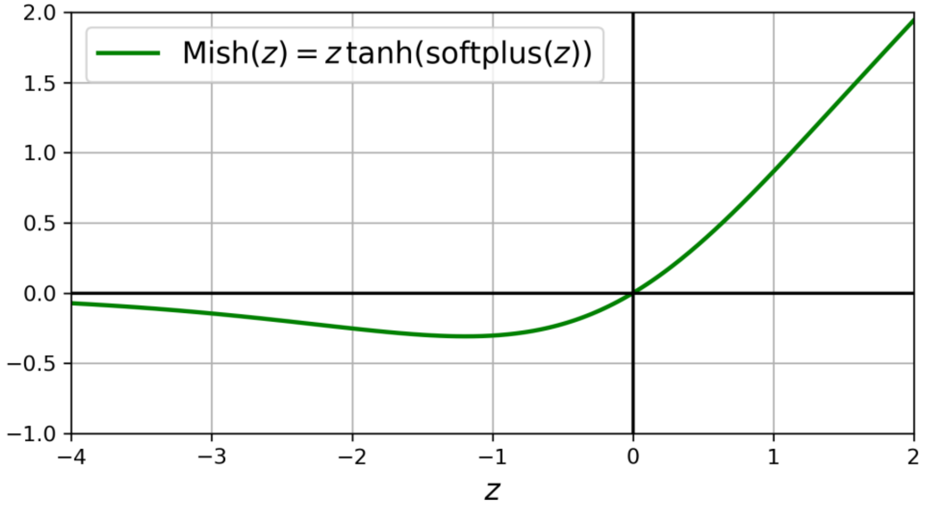
$$GELU(z) \approx Swish_{1.702}(z)$$

- You can tune β like any other hyperparameter.
- It's possible to make β trainable and gradient descent optimizes it.
- Like PReLU, this can make your model more powerful, but it also runs the risk of overfitting.



Mish

- *Mish* is introduced in a 2019 paper by Diganta Misra and is defined as $mish(z) = z.\,tanh(softplus(z))$, where $softplus(z) = \log(1 + \mathrm{e}^{z})$.
- Like GELU and Swish, it is a smooth, nonconvex, and nonmonotonic variant of ReLU.



Which activation function should we use?

- ReLU remains a good default for simple tasks:
  - It’s often just as good as the more sophisticated activation functions
  - It’s very fast to compute, and many libraries and hardware accelerators provide ReLU-specific optimizations.
- Swish is a better default for more complex tasks, and you can even try parametrized Swish with a learnable β parameter for the most complex tasks.
  - Mish gives you slightly better results, but it requires a bit more compute.
- If you care a lot about runtime latency, then you may prefer leaky ReLU, or parametrized leaky ReLU for more complex tasks.
- For deep MLPs, use SELU, but make sure to respect the constraints.

Batch Normalization

- *Batch Normalization* technique consists of adding an operation in the model just before or after the activation function of each hidden layer.
- This operation zero-centers and normalizes each input, then scales and shifts the result using two new learnable parameter vectors per layer: $\boldsymbol{\gamma}$ for scaling, and $\boldsymbol{\beta}$ for shifting.
  - The operation lets the model learn the optimal scale and mean of each of the layer's inputs.
- In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation.
  - It does so by evaluating the mean and standard deviation of the input over the current mini-batch.

Batch Normalization

- $\boldsymbol{\mu}_B$: vector of input means over B.
- m_B: number of instances in batch B.
- $\boldsymbol{\sigma}_B$: vector of input standard deviations.
- $\hat{\mathbf{x}}^{(i)}$: normalized inputs for instance i.
- ε: smoothing term (typically 10^{-5}).
- $\boldsymbol{\gamma}$: the output scale parameter vector.
- $\otimes$: element-wise multiplication
- $\boldsymbol{\beta}$: the output shift parameter vector.
- $\boldsymbol{z}^{(i)}$: the output of the BN operation.

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. $$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B{}^2 + \varepsilon}}$$
4. $$\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

Batch Normalization at Test Time

- During training, BN standardizes its inputs, then rescales and offsets them. What about at test time?
  - We may need to make predictions for individual instances rather than for batches of instances.
  - We have no way to compute each input’s mean and standard deviation.
  - Even if we do have a batch of instances, it may be too small, or the instances may not be independent and identically distributed.
- Most implementations of BN estimate the final statistics during training by using a moving average of the layer’s input means and standard deviations.
  - This is what Keras does automatically when you use the `BatchNormalization` layer.

Batch Normalization

- Four parameter vectors are learned in each batch-normalized layer:
  - $\boldsymbol{\gamma}$ (the output scale vector) and $\boldsymbol{\beta}$ (the output offset vector) are learned through regular backpropagation.
  - $\boldsymbol{\mu}$ (the final input mean vector) and $\boldsymbol{\sigma}$ (the final input standard deviation vector) are estimated using an exponential moving average.

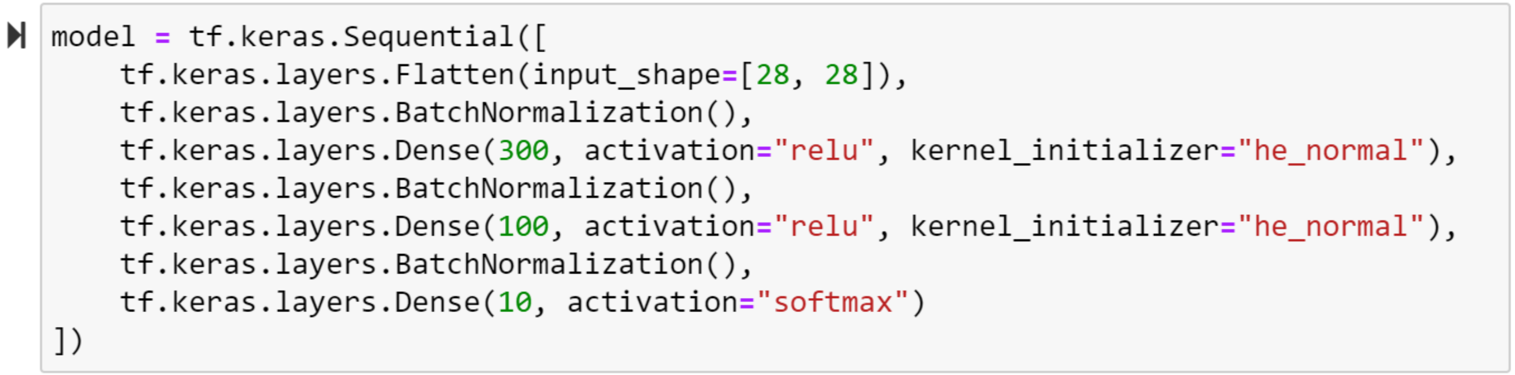
- BN considerably improves most deep neural networks:
  - The vanishing gradients problem is strongly reduced, and you could even use saturating activation functions such as the tanh and sigmoid.
  - The networks are much less sensitive to the weight initialization.
  - You can use much larger learning rates, significantly speeding up the learning process.
  - BN also acts like a regularizer, preventing overfitting.

Disadvantage of Batch Normalization

- BN adds some complexity to the model.
- There is also a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer.
- It is often possible to fuse the BN layer with the previous layer after training, thereby avoiding the runtime penalty:
  - If the previous layer computes $\boldsymbol{XW} + \boldsymbol{b}$, then the BN layer will compute $\boldsymbol{\gamma} \otimes (\boldsymbol{XW} + \boldsymbol{b} - \boldsymbol{\mu})/\boldsymbol{\sigma} + \boldsymbol{\beta}$
  - If we define $\boldsymbol{W}' = \boldsymbol{\gamma} \otimes \boldsymbol{W}/\boldsymbol{\sigma}$ and $\boldsymbol{b}' = \boldsymbol{\gamma} \otimes (\boldsymbol{b} - \boldsymbol{\mu})/\boldsymbol{\sigma} + \boldsymbol{\beta}$, the equation simplifies to $\boldsymbol{XW}' + \boldsymbol{b}'$.
  - We can replace the previous layer's weights and biases ($\boldsymbol{W}$ and $\boldsymbol{b}$) with $\boldsymbol{W}'$ and $\boldsymbol{b}'$, and get rid of the BN layer.

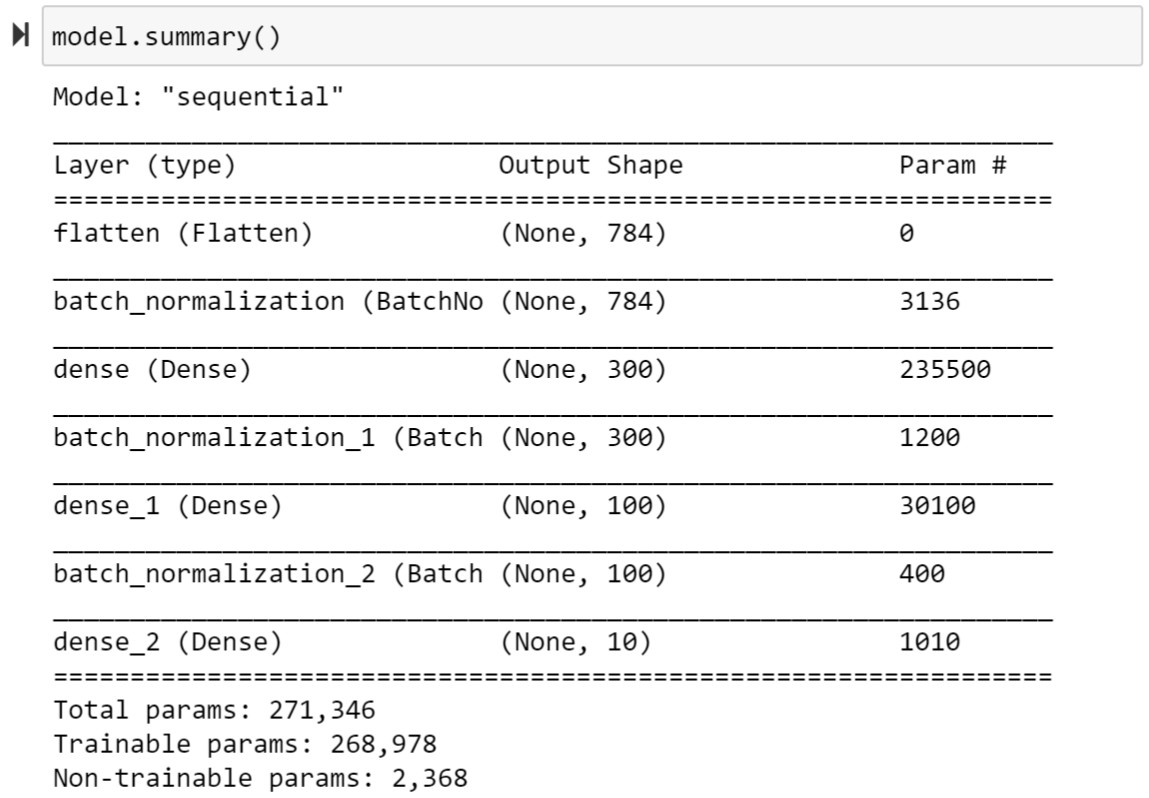
Implementing Batch Normalization

- Just add a `BatchNormalization` layer before or after each hidden layer's activation function.
- You may add a BN layer as the first layer in your model, but a plain `Normalization` layer generally performs just as well.
  - The drawback of plain normalization is that you must first call its `adapt()` method.



```
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")
])
```

Model Summary



```
model.summary()
```

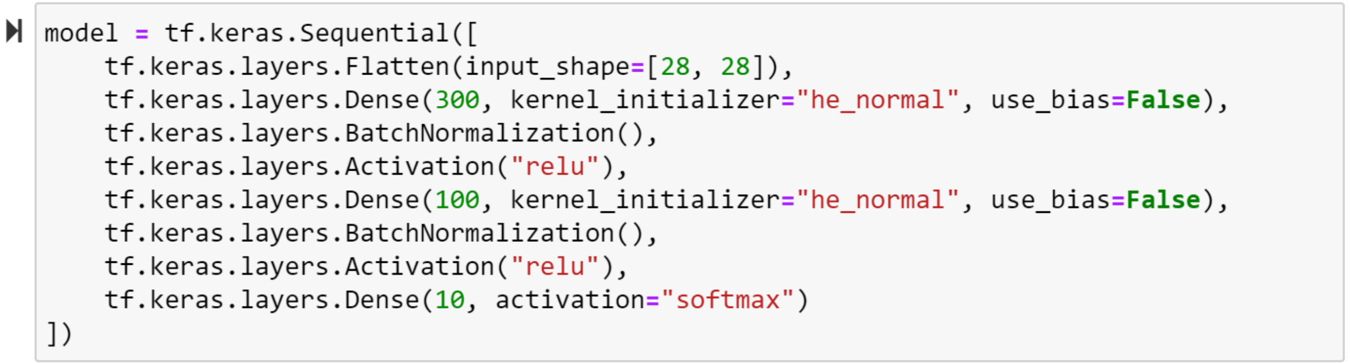
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten (Flatten)            (None, 784)               0         
_________________________________________________________________
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
Trainable params: 268,978
Non-trainable params: 2,368
```

Implementing Batch Normalization

- The authors of the BN paper argued in favor of adding the BN layers before the activation functions.
- To add the BN layers before the activation function, you must remove the activation functions from the hidden layers and add them as separate layers after the BN layers.
- Since a batch normalization layer includes one offset parameter per input, you can remove the bias term from the previous layer by passing `use_bias=False` when creating it.
- You can usually drop the first BN layer to avoid sandwiching the first hidden layer between two BN layers.

Implementing Batch Normalization

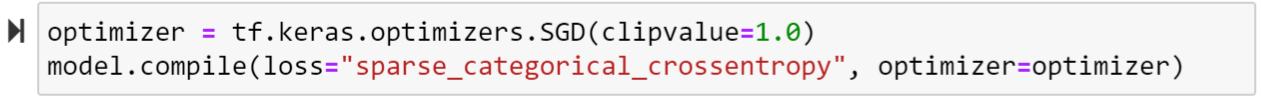
- The `BatchNormalization` class has quite a few hyperparameters you can tweak. The defaults will usually be fine.
- You may occasionally need to tweak the `momentum`.
  - It is used by the `BatchNormalization` layer when it updates the exponential moving averages.
- Batch normalization has become one of the most-used layers in deep neural networks, especially deep convolutional neural networks.



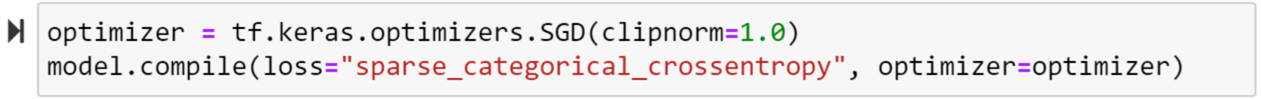
```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.Dense(100, kernel_initializer="he_normal",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

Gradient Clipping

- *Gradient Clipping*: to mitigate the exploding gradients problem, clip the gradients during backpropagation so that they never exceed some threshold.
  - This technique is generally used in recurrent neural networks, where using batch normalization is tricky.
- In Keras, implement gradient clipping by setting the `clipvalue` or `clipnorm` argument when creating an optimizer:



```
optimizer = tf.keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer)
```



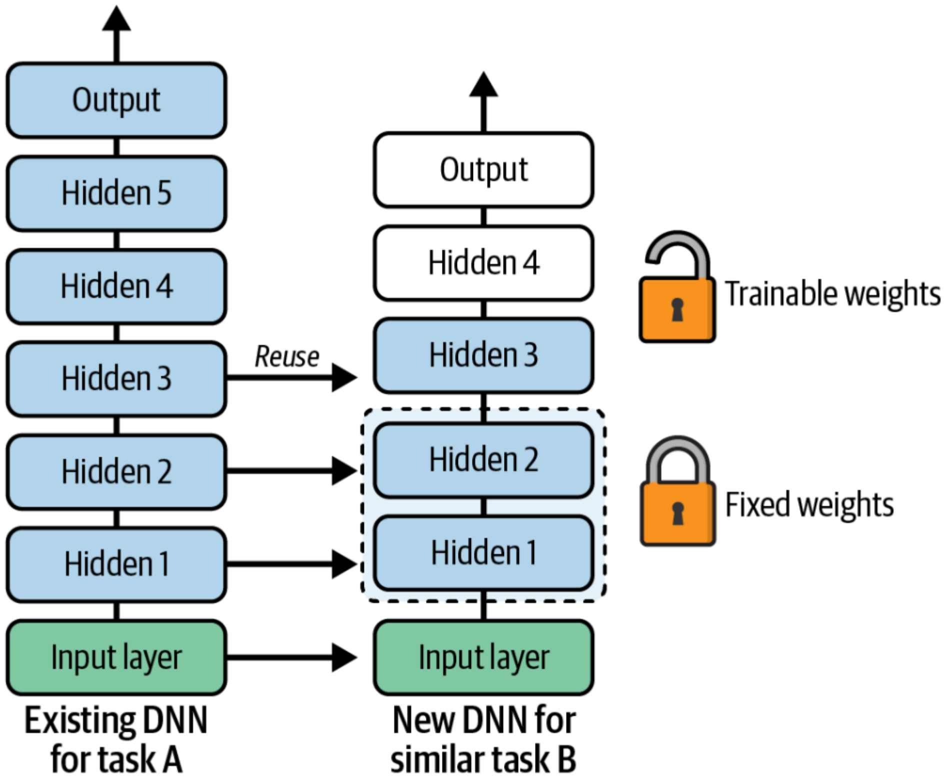
```
optimizer = tf.keras.optimizers.SGD(clipnorm=1.0)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer)
```

Gradient Clipping

- Setting `clipvalue=1.0` optimizer will clip every component of the gradient vector to a value between –1.0 and 1.0.
- Note that it may change the orientation of the gradient vector.
  - E.g. the gradient vector [0.9, 100.0] points mostly in the direction of the second axis; but once you clip it by value, you get [0.9, 1.0], which points roughly at the diagonal between the two axes.
- To ensure that gradient clipping does not change the direction of the gradient vector, clip by norm using `clipnorm` instead of `clipvalue`.
  - This will clip the whole gradient if its ℓ_2 norm is greater than the threshold you picked.

2.
Reusing Pretrained Layers

Transfer Learning



Transfer Learning

- Finding the right number of layers to reuse is important.
- The more similar the tasks are, the more layers you will want to reuse (starting with the lower layers).
  - For very similar tasks, try to keep all the hidden layers and just replace the output layer.
- Freeze reused layers, then train your model and see how it performs.
  - Try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves.
  - The more training data you have, the more layers you can unfreeze.
- It is useful to reduce the learning rate when you unfreeze reused layers to avoid wrecking their fine-tuned weights.

Transfer Learning in Keras

- Let's split the fashion MNIST training set in two:
  - `X_train_A`: all images of all items except for T-shirts/tops and pullovers.
  - `X_train_B`: a much smaller training set of just the first 200 images of T-shirts/tops and pullovers.
- We will train a model on set A (classification task with 8 classes), and try to reuse it to tackle set B (binary classification).
  - We hope to transfer a little bit of knowledge from task A to task B, since classes in set A are somewhat similar to classes in set B.
- Since we are using `Dense` layers, only patterns that occur at the same location can be reused.
  - Convolutional layers will transfer much better, since learned patterns can be detected anywhere on the image.

Transfer Learning in Keras

```python
model_A = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dense(8, activation="softmax")
])

model_A.compile(loss="sparse_categorical_crossentropy",
                optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
                metrics=["accuracy"])
history = model_A.fit(X_train_A, y_train_A, epochs=20,
                      validation_data=(X_valid_A, y_valid_A))
model_A.save("my_model_A")
```

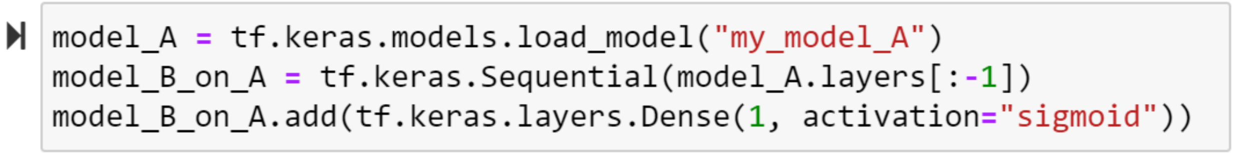
```python
model_B = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

model_B.compile(loss="binary_crossentropy",
                optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
                metrics=["accuracy"])
history = model_B.fit(X_train_B, y_train_B, epochs=20,
                      validation_data=(X_valid_B, y_valid_B))
model_B.evaluate(X_test_B, y_test_B)
```

➢ Model B reaches 91.85% accuracy on the test set.

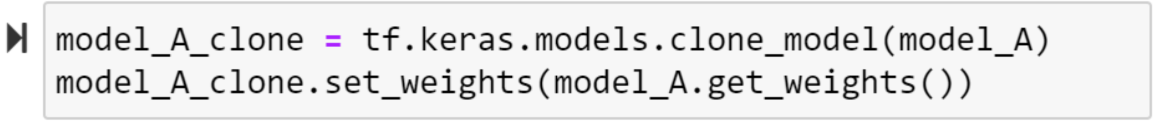
Transfer Learning in Keras

- To reuse model A, you need to load it and create a new model based on that model's layers. Here we reuse all the layers except for the output layer:



```
model_A = tf.keras.models.load_model("my_model_A")
model_B_on_A = tf.keras.Sequential(model_A.layers[:-1])
model_B_on_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

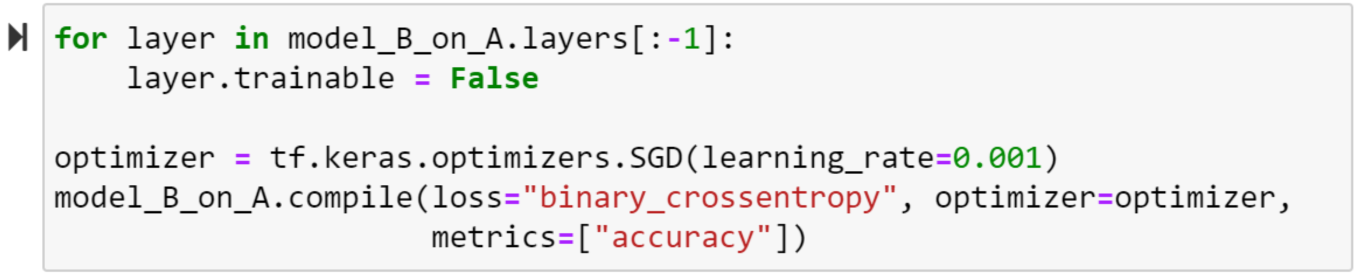
- Note that `model_A` and `model_B_on_A` now share some layers. When you train `model_B_on_A`, it will also affect `model_A`.
- To avoid that, you need to *clone* `model_A` before you reuse its layers:



```
model_A_clone = tf.keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Transfer Learning in Keras

- Since the new output layer is initialized randomly, it will make large errors and the large error gradients may wreck the reused weights.
- To avoid this, we can freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights.
- To do this, set every layer’s `trainable` attribute to `False` and compile the model:

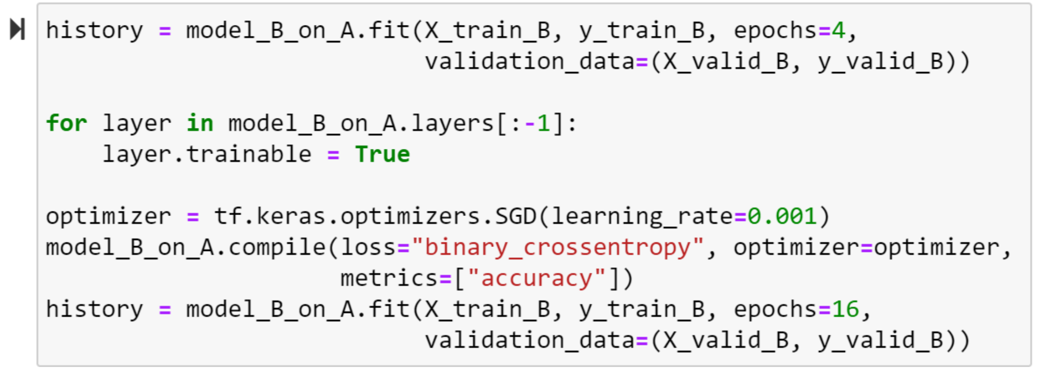


```python
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
```

Transfer Learning in Keras

- Train the model for a few epochs, then unfreeze the reused layers (which requires compiling the model again) and continue training to fine-tune the reused layers for task B.
- After unfreezing the reused layers, it is usually a good idea to reduce the learning rate, to avoid damaging the reused weights.



```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

- This model’s test accuracy is 93.85%.

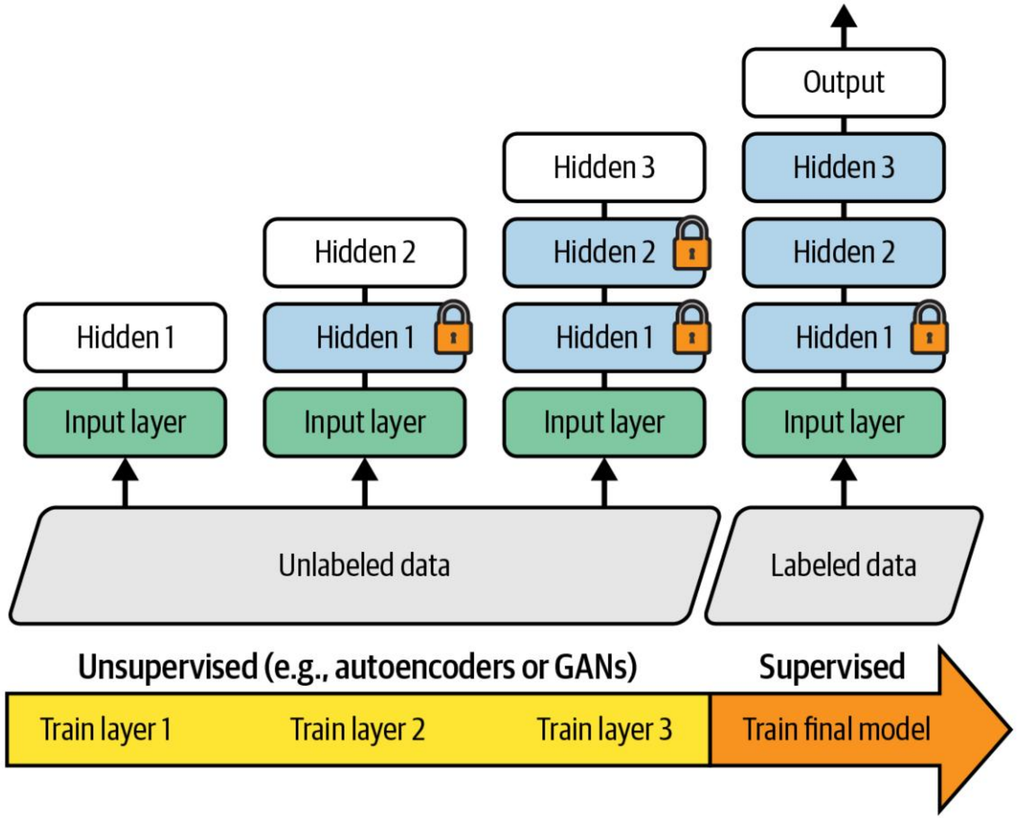
Unsupervised Pretraining

- We have a complex task for which we don’t have much labeled training data, and we cannot find a model trained on a similar task.
  - It is often cheap to gather unlabeled training examples, but expensive to label them.
- If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model, such as an autoencoder or a generative adversarial network.
- You can reuse the lower layers of the autoencoder or the lower layers of the GAN’s discriminator, add the output layer for your task on top, and fine-tune the final network using supervised learning.

Unsupervised Pretraining

- It is this technique that Geoffrey Hinton and his team used in 2006, and which led to the revival of neural networks.
- Until 2010, unsupervised pretraining (using restricted Boltzmann machines) was the norm for deep nets, and only after the vanishing gradients problem was reduced, it became more common to train DNNs purely using supervised learning.
- Unsupervised pretraining (today typically using autoencoders or GANs rather than RBMs) is still a good option when:
  - you have a complex task to solve
  - no similar model you can reuse
  - little labeled training data but plenty of unlabeled training data

Unsupervised Pretraining



Pretraining on an Auxiliary Task

- If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task.
- Example: if you want to build a system to recognize faces, you may only have a few pictures of each individual.
  - You could gather a lot of pictures of random people on the web and train a first neural network to detect whether or not two different pictures feature the same person.
  - Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier that uses little training data.