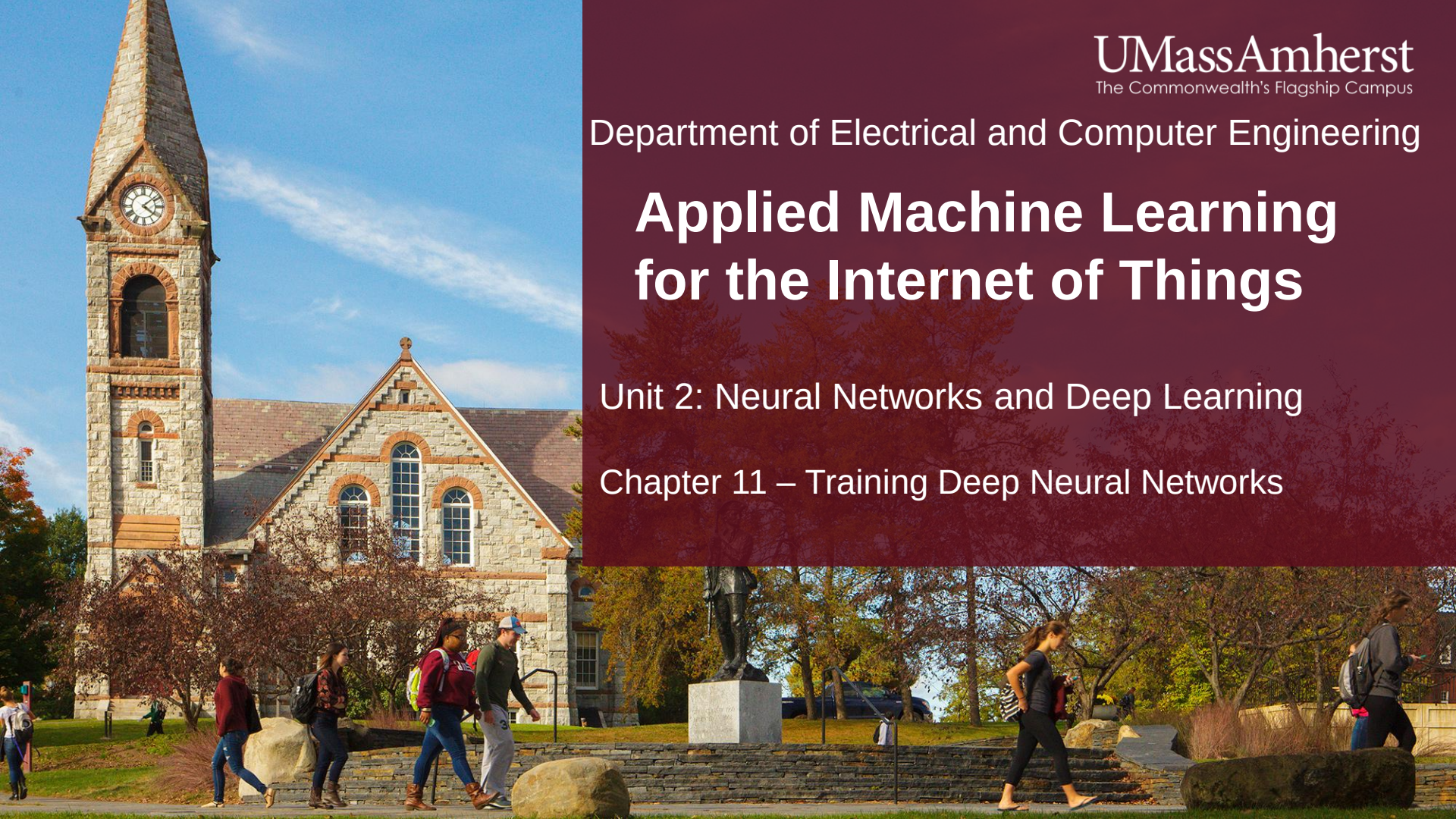


UMassAmherst
The Commonwealth's Flagship Campus

Department of Electrical and Computer Engineering

Applied Machine Learning for the Internet of Things

Unit 2: Neural Networks and Deep Learning

Chapter 11 – Training Deep Neural Networks

Rationale

- Artificial neural networks can have many layers
  - Deep neural networks for complex problems
- Challenges when training deep neural networks
  - Vanishing/exploding gradient during backward propagation
  - Insufficient data for large network
  - Training may be extremely slow
  - Risk of overfitting model with millions of parameters
- We consider these challenges and discuss solutions
  - Widely used approaches in deep learning

Objectives

- Vanishing/exploding gradients problems
- Reusing pretrained layers
- Faster Optimizers
- Avoiding overfitting through regularization

Prior Knowledge

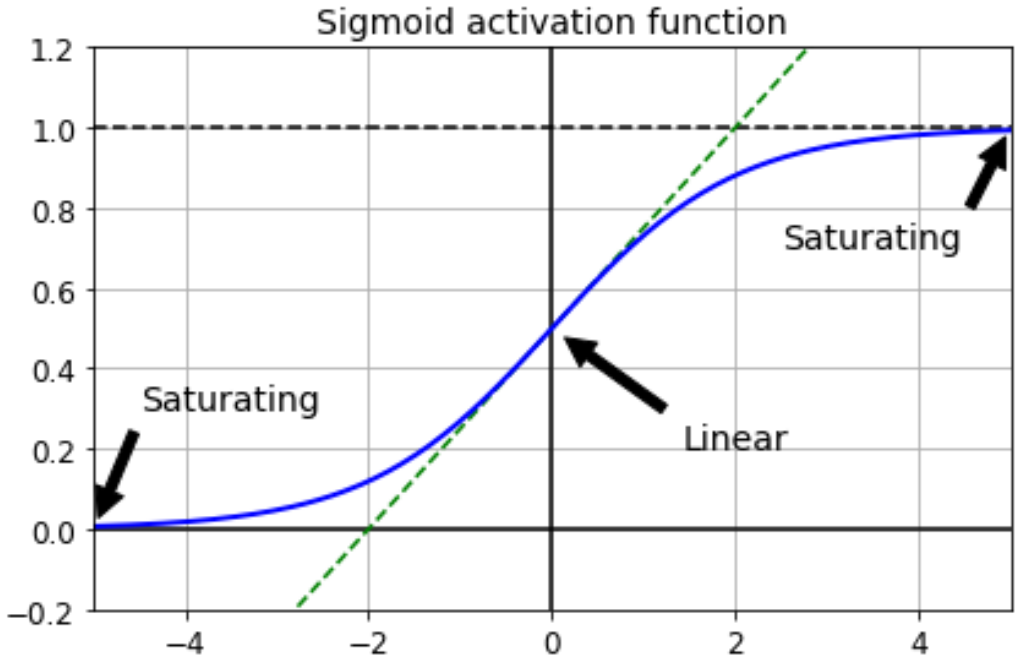
- A general understanding of computer science or computer engineering.
- A general experience with programming (e.g., Python).
- A general knowledge of linear algebra.

Vanishing/Exploding Gradients Problem

- Backpropagation going from output layer back to input layer
  - Propagating error gradient backwards
- Gradients often get smaller as algorithm propagates back/down
  - Gradient Descent update leaves connection weights virtually unchanged
  - “Vanishing gradients problem”
- In some cases, the opposite occurs
  - Gradients increase as algorithm diverges
  - “Exploding gradients problem”
- Problem lies in sigmoid activation function and type of initialization
  - Variance of output of layer is much higher than variance of input

Vanishing/Exploding Gradients Problem

- Logistic/sigmoid function saturates at 0 or 1
  - Nearly no gradient at that point
- Very limited gradient
  - Even less when back-propagating



Glorot and He Initialization

- Aim is to maintain “signal” when going through layers
  - Input variance equal to output variance of layer
  - Helpful for forward and backward direction
- Not possible under all circumstances, but initialization can help
- Glorot initialization based on average fan-in and fan-out of layer

Equation 11-1. Glorot initialization (when using the logistic activation function)

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

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

Glorot and He Initialization

- LeCun and He proposed similar initializations
  - Different recommendations depending on activation functions

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

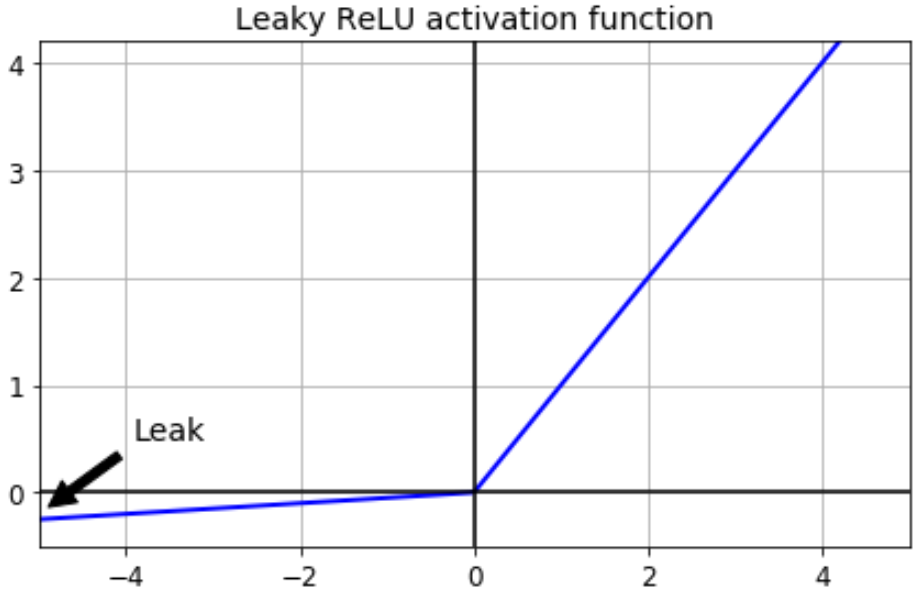
Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Nonsaturating Activation Functions

- Unstable gradients in part due to poor choice of activation function
  - Logistic/sigmoid activation occurs in biological neural networks
  - Deep neural networks better with different functions
- ReLU (Rectified Linear Unit) good, but not perfect
  - Suffers from “dying ReLU” problem
  - Neurons stop outputting anything but zero
  - Gradient Descent doesn’t work well anymore
- Variant of ReLU: leaky ReLU
  - $\text{LeakyReLU}_{\alpha}(z) = \max(\alpha z, z)$
  - Maintains small slope in negative range
  - Typical value for α is 0.01

Leaky ReLU

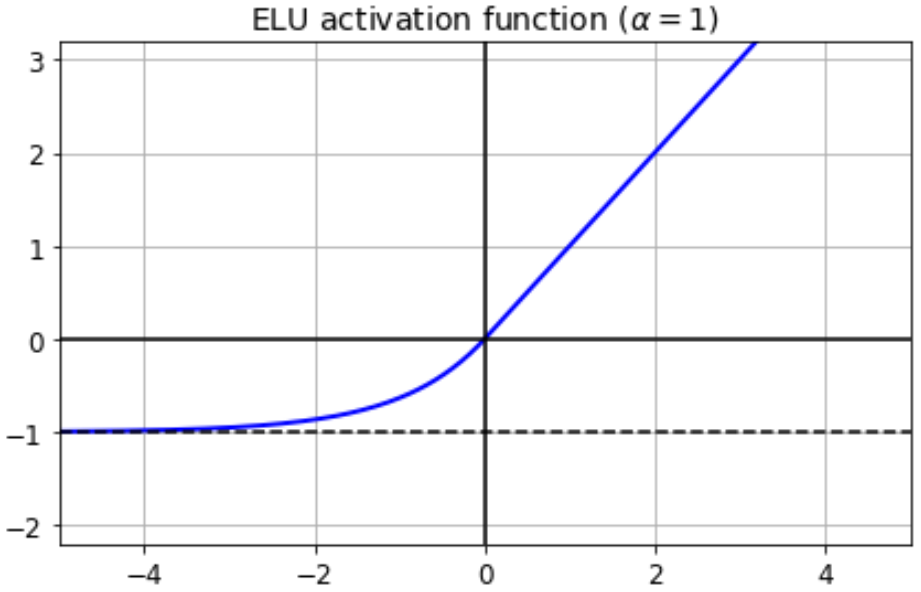
- Leaky ReLU:



- Variants:
  - Randomized leaky ReLU (RReLU): random choice of parameter
  - Parametric leaky ReLU (PReLU): parameter can be learned during training

Exponential Linear Unit

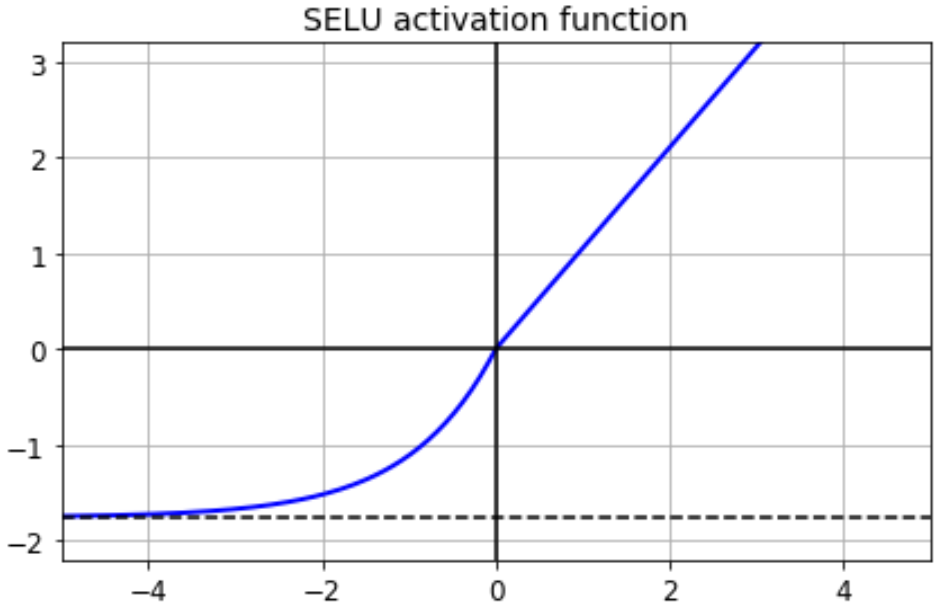
- Exponential linear unit (ELU)
  - $\mathrm{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$
  - Helps with faster training despite of higher computational cost



- ELU characteristics
  - Average close to 0 to avoid vanishing gradients
  - Nonzero gradient in negative range to avoid dead neurons problem
  - For $\alpha = 1$ smooth everywhere to avoid bouncing near 0

Scaled ELU

- Scaled ELU (SELU)
  - $\text{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leqslant 0 \end{cases}$



- SELU has self-normalizing property for stack of dense layers
  - Input features must be standardized (mean 0 and standard deviation 1)
  - Hidden layers' weights need to be initialized with LeCun normal initialization
  - Network needs to be sequential

Batch Normalization

- He initialization and ELU can reduce vanishing/exploding gradients
  - May come back during training
- Batch normalization (BN) addresses problem
  - Operation before and after activation function of layer
    - Operation zero-centers and normalizes each input
    - Operation scales and shifts results using two parameter vectors per layer
- Batch normalization layer at input replaces StandardScaler

Batch Normalization

- Batch Normalization algorithm
  - μ_B is vector of input means over mini-batch B (one per input)
  - σ_B is vector of input standard deviations over mini-batch B (one per input)
  - m_B is number of instances in mini-batch B
  - $\hat{x}^{(i)}$ is vector of zero-centered and normalized inputs for instance i
  - $z^{(i)}$ is vector of rescaled and shifted of inputs
    - γ is output scale vector (learned)
    - β is output offset vector (learned)

Equation 11-3. Batch Normalization algorithm

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

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

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

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

Batch Normalization

- BN layer is easy to add in Keras

```
In [35]: model = keras.models.Sequential([
             keras.layers.Flatten(input_shape=[28, 28]),
             keras.layers.BatchNormalization(),
             keras.layers.Dense(300, activation="relu"),
             keras.layers.BatchNormalization(),
             keras.layers.Dense(100, activation="relu"),
             keras.layers.BatchNormalization(),
             keras.layers.Dense(10, activation="softmax")
         ])
```

  - Adds additional variables to model

```
In [37]: bn1 = model.layers[1]
         [(var.name, var.trainable) for var in bn1.variables]

Out[37]: [('batch_normalization/gamma:0', True),
          ('batch_normalization/beta:0', True),
          ('batch_normalization/moving_mean:0', False),
          ('batch_normalization/moving_variance:0', False)]
```

Batch Normalization

- Batch normalization has hyperparameters
  - Momentum for exponential moving averages

$$\hat{\mathbf{v}} \leftarrow \hat{\mathbf{v}} \times \text{momentum} + \mathbf{v} \times (1 - \text{momentum})$$

    - Good momentum value close to 1 (e.g., 0.9, 0.99, 0.999)
  - Axis determines which axis should be normalized
    - Default -1 (last axis using mean and stddev across all axes)
- Batch Normalization has become one of the most-used layers
  - Widely used in deep neural networks
  - Often omitted from diagrams (since it is simply assumed to be present)

Gradient Clipping

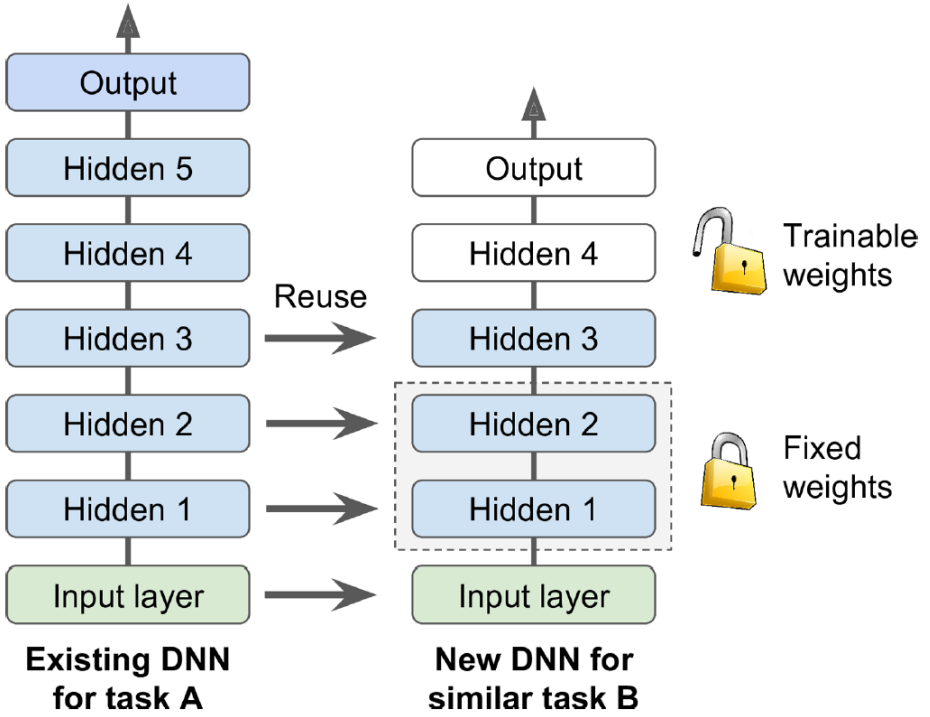
- Another technique to mitigate exploding gradients problem
  - Clip gradients during backpropagation
  - Often used in recurrent neural networks instead of batch normalization
- Clipping in range [-1..1]
  - Clipping parameter is hyperparameter
  - Clipping may change direction of vectors
- Clipping by ℓ_2 norm preserves direction of vector

Reusing Pretrained Layers

- Training a large DNN from scratch is difficult
  - Need lots of data training data and training time
- Reuse of trained networks is possible: transfer learning
  - Reuse lower layers of neural network trained on similar task
- Example:
  - Have DNN trained to classify 100 categories of animals, plants, vehicles, objects
  - Need DNN to classify different types of vehicles
  - Reuse lower layers of DNN and train additional layers
    - Upper layers less likely to be useful since they differ depending on task
    - Outputs may be different

Transfer Learning

- Illustration of layer reuse
- Lower layer weights are locked
  - Protects their fine-tuned weights
- If results are not good
  - Change number of top layers
  - Unlock some layers' weights after some initial training



Transfer Learning in Keras

- Example assume Fashion MNIST has only eight classes
  - Sandal and Shirt classes removed
- Model A: train Keras model on eight Fashion MNIST classes
  - Use full training set for those eight classes and achieve good accuracy (>90%)
- Model B: binary classifier for shirt (positive) and sandals (negative)
  - Limited training set of only 200 images
- Option 1: train Keras classifier directly based on 200 images
  - Achieve 97.2% accuracy
- Option 2: transfer some layers from Model A, then train
  - Achieve 99.25% accuracy

Transfer Learning in Keras

- Split dataset:

```
In [46]: def split_dataset(X, y):
             y_5_or_6 = (y == 5) | (y == 6) # sandals or shirts
             y_A = y[~y_5_or_6]
             y_A[y_A > 6] -= 2 # class indices 7, 8, 9 should be moved to 5, 6, 7
             y_B = (y[y_5_or_6] == 6).astype(np.float32) # binary classification task: is it a shirt (cl
             return ((X[~y_5_or_6], y_A),
                     (X[y_5_or_6], y_B))

         (X_train_A, y_train_A), (X_train_B, y_train_B) = split_dataset(X_train, y_train)
         (X_valid_A, y_valid_A), (X_valid_B, y_valid_B) = split_dataset(X_valid, y_valid)
         (X_test_A, y_test_A), (X_test_B, y_test_B) = split_dataset(X_test, y_test)
         X_train_B = X_train_B[:200]
         y_train_B = y_train_B[:200]

In [47]: X_train_A.shape

Out[47]: (43986, 28, 28)

In [48]: X_train_B.shape

Out[48]: (200, 28, 28)
```

Transfer Learning in Keras

- Model A (92.1% accuracy):

```
In [52]: model_A = keras.models.Sequential()
         model_A.add(keras.layers.Flatten(input_shape=[28, 28]))
         for n_hidden in (300, 100, 50, 50, 50):
             model_A.add(keras.layers.Dense(n_hidden, activation="selu"))
         model_A.add(keras.layers.Dense(8, activation="softmax"))

In [53]: model_A.compile(loss="sparse_categorical_crossentropy",
                         optimizer=keras.optimizers.SGD(lr=1e-3),
                         metrics=["accuracy"])

In [54]: history = model_A.fit(X_train_A, y_train_A, epochs=20,
                               validation_data=(X_valid_A, y_valid_A))
```

```
         val_loss: 0.2330 - val_accuracy: 0.9208
         Epoch 20/20
         1375/1375 [==============================] - 4s 3ms/step - loss: 0.2156 - accuracy: 0.9261 -
         val_loss: 0.2332 - val_accuracy: 0.9205

In [55]: model_A.save("my_model_A.h5")
```

Transfer Learning in Keras

- Model B – Option 1 (97.2% accuracy):

```
In [56]: model_B = keras.models.Sequential()
         model_B.add(keras.layers.Flatten(input_shape=[28, 28]))
         for n_hidden in (300, 100, 50, 50, 50):
             model_B.add(keras.layers.Dense(n_hidden, activation="selu"))
         model_B.add(keras.layers.Dense(1, activation="sigmoid"))

In [57]: model_B.compile(loss="binary_crossentropy",
                         optimizer=keras.optimizers.SGD(lr=1e-3),
                         metrics=["accuracy"])

In [58]: history = model_B.fit(X_train_B, y_train_B, epochs=20,
                               validation_data=(X_valid_B, y_valid_B))
```

```
oss: 0.1482 - val_accuracy: 0.9716
Epoch 20/20
7/7 [==============================] - 0s 12ms/step - loss: 0.1208 - accuracy: 0.9900 - val_l
oss: 0.1431 - val_accuracy: 0.9716
```

Transfer Learning in Keras

- Model B – Option 2 model construction
  - Load layers from Model A, add layer, clone, lock weights on transferred layers

In [60]:
```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

In [61]:
```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

In [62]:
```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy",
                     optimizer=keras.optimizers.SGD(lr=1e-3),
                     metrics=["accuracy"])
```

Transfer Learning in Keras

- Model B – Option 2 model training (train four epochs, then unlock layers)

```
In [63]: 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

         model_B_on_A.compile(loss="binary_crossentropy",
                              optimizer=keras.optimizers.SGD(lr=1e-3),
                              metrics=["accuracy"])
         history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                                   validation_data=(X_valid_B, y_valid_B))

         Epoch 1/4
         7/7 [==============================] - 0s 35ms/step - loss: 0.5776 - accuracy: 0.6550 - val_l
         oss: 0.5824 - val_accuracy: 0.6359
         Epoch 2/4
         7/7 [==============================] - 0s 12ms/step - loss: 0.5412 - accuracy: 0.6800 - val_l
         oss: 0.5451 - val_accuracy: 0.6836
         Epoch 3/4
         7/7 [==============================] - 0s 11ms/step - loss: 0.5046 - accuracy: 0.7250 - val_l
         oss: 0.5131 - val_accuracy: 0.7099
         Epoch 4/4
         7/7 [==============================] - 0s 12ms/step - loss: 0.4732 - accuracy: 0.7450 - val_l
         oss: 0.4845 - val_accuracy: 0.7343
         Epoch 1/16
         7/7 [==============================] - 0s 36ms/step - loss: 0.3952 - accuracy: 0.8150 - val_l
```

Transfer Learning in Keras

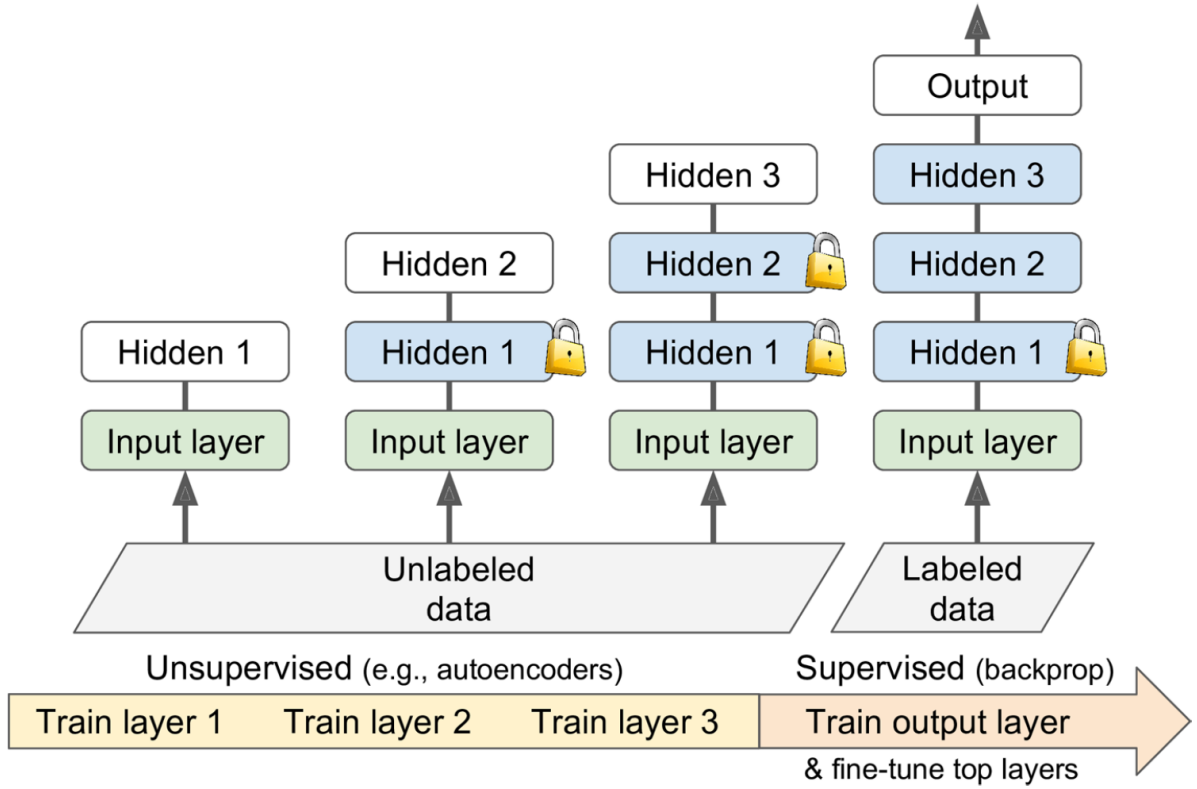
- Comparison of Model B options

```
In [64]: model_B.evaluate(X_test_B, y_test_B)

         63/63 [==============================] - 0s 2ms/step - loss: 0.1408 - accuracy: 0.9705

Out[64]: [0.1408407837152481, 0.9704999923706055]

In [65]: model_B_on_A.evaluate(X_test_B, y_test_B)

         63/63 [==============================] - 0s 2ms/step - loss: 0.0683 - accuracy: 0.9935

Out[65]: [0.06827671080827713, 0.9934999942779541]
```

  - Transfer learning outperforms training from scratch
- Improvement depends on specific parameters
  - Limited use for dense networks; better for convolutional neural networks

Unsupervised Pretraining

- Training with unlabeled data
  - Use few labeled instances later
- Layer-by-layer training
  - Greedy layer-wise pretraining



Faster Optimizers

- Training for very deep neural networks can be very slow
- Strategies for speeding up training
  - Applying good initialization of weights
  - Using good activation function
  - Using Batch Normalization
  - Reusing part of a pretrained network built on a similar task
- Additional opportunity for speeding up training: optimizer
  - Use faster optimizer than Gradient Descent optimizer
  - Examples: momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, Adam and Nadam optimization

Momentum Optimization

- Momentum optimizer "picks up speed" when descending
  - Momentum gain based on gradient (gradient is acceleration, not speed)

Equation 11-4. Momentum algorithm

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

- Hyperparameter β called momentum
  - Ranging from 0 (high friction) to 1 (no friction)

```
In [67]: optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Nesterov Accelerated Gradient

- Variation on Momentum Optimization
  - Measures gradient of cost function slightly ahead in the direction of momentum

Equation 11-5. Nesterov Accelerated Gradient algorithm

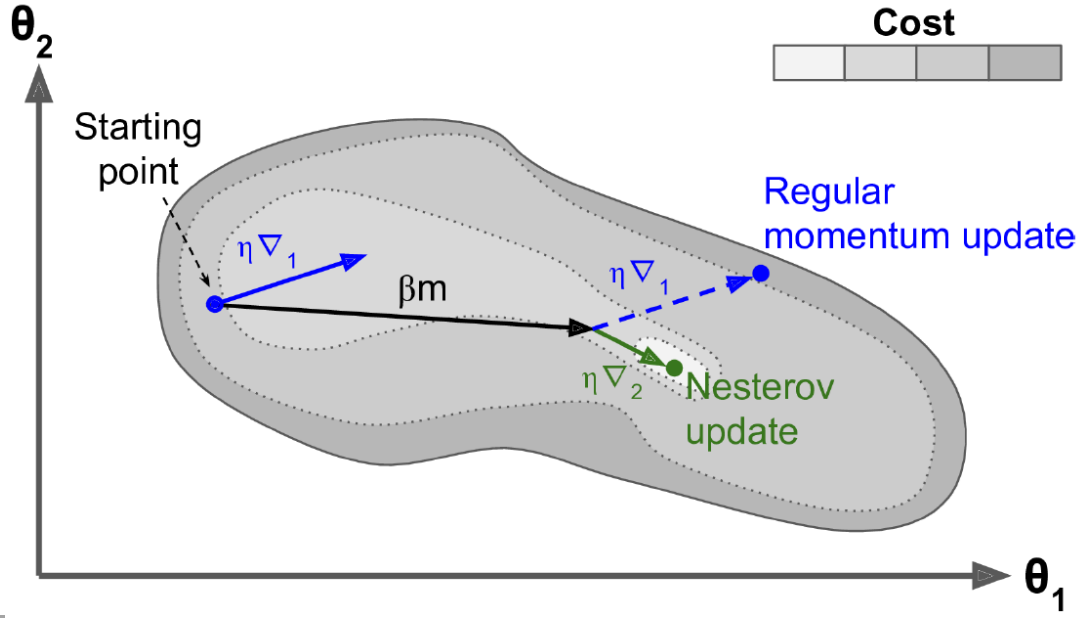
1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta\mathbf{m})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

  - Generally faster than plain Momentum Optimization

```
In [68]: optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

Comparison of Momentum Optimizers

- Momentum Optimization vs. Nesterov Accelerated Gradient
  - NAG does better by taking the gradient at end of black arrow instead of beginning

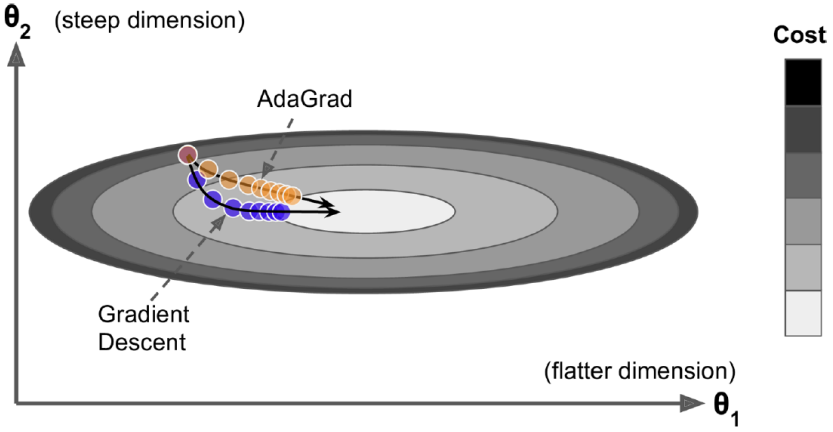


AdaGrad

- AdaGrad scales gradient vector along steepest dimension
  - Avoids misdirection in an optimization "bowl"

Equation 11-6. AdaGrad algorithm

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$



  - Works well for simple quadratic problems
  - But may stop before global optimum
  - Does not work well for deep neural networks

```
In [69]: optimizer = keras.optimizers.Adagrad(lr=0.001)
```

RMSProp

- RMSProp addresses risk of AdaGrad to stop too early
  - Only accumulates gradients from most recent iterations with exponential decay

Equation 11-7. RMSProp algorithm

1. $\mathbf{s} \leftarrow \beta\mathbf{s} + (1-\beta)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\, \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$

  - Typical decay rate of 0.9

```
In [70]: optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Adam and Nadam Optimization

- Adaptive Moment Estimation (Adam)
  - Similar to Moment Optimization
    - Keeps exponentially decaying average of past gradients
  - Similar to RMSProp
    - Keeps exponentially decaying average of past squared gradients
  - Steps 3 & 4 correct bias toward zero from initialization
- Nadam is Adam plus Nesterov trick

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1 - \beta_1^{\,t}}$
4. $\widehat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1 - \beta_2^{\,t}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon}$

```
In [71]: optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Other Optimizers

- Previous examples rely on first-order partial derivatives (Jacobians)
  - Literature also includes optimizer based on second-order partial derivatives
    - Hessian (partial derivatives of Jacobians)
- Hessians difficult for deep neural networks
  - n^2 Hessians per output compared to n Jacobians

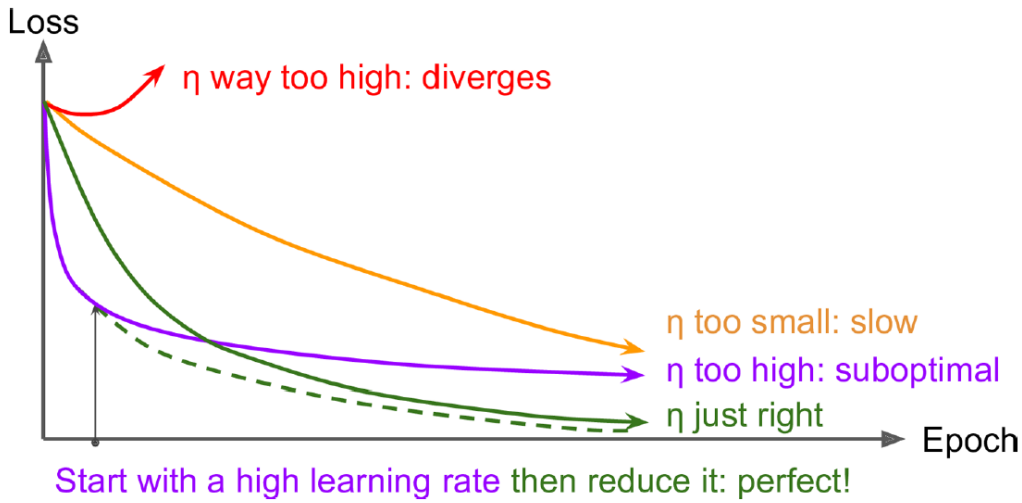
Optimizer Choice

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 ***
Nadam	***	** or ***
AdaMax	***	** or ***

Learning Rate Schedule

- Learning rate is important
  - Low learning rate:
    - Finds the optimum
    - Training takes long
  - High learning rate:
    - Training may diverge
    - Training may not settle down
- Best of both worlds
  - Start with high rate, then reduce



Learning Rate Schedule

- Learning rate schedules
  - Power scheduling
  - Exponential scheduling
  - Piecewise constant scheduling
  - Performance scheduling
  - Others (e.g., 1cycle)
- Learning rate schedulers update the optimizer's learning rate
  - At the beginning of epoch
  - Via parameter in optimizer or through callback function

Learning Rate Schedule

- Example of power scheduling (parameter):

```
In [74]: optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)

In [75]: model = keras.models.Sequential([
             keras.layers.Flatten(input_shape=[28, 28]),
             keras.layers.Dense(300, activation="selu", kernel_initializer="lecun_normal"),
             keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal"),
             keras.layers.Dense(10, activation="softmax")
         ])
         model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"]
```

Learning Rate Schedule

- Example of exponential scheduling (callback):

In [78]:
```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

In [79]:
```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

In [80]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="selu", kernel_initializer="lecun_normal"),
    keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal"),
    keras.layers.Dense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accuracy"])
n_epochs = 25
```

In [81]:
```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbacks=[lr_scheduler])
```

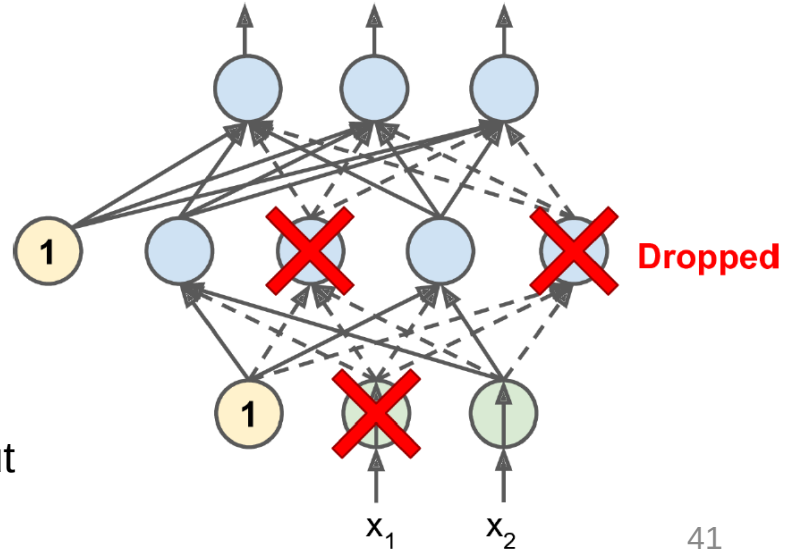
Regularization

- Avoid overfitting through regularization
  - Deep neural networks have thousands of parameters, thus prone to overfitting
- Batch Normalization acts as regularization
- ℓ_2 and ℓ_1 regularization
  - ℓ_2 regularization constrains neural network weights
  - ℓ_1 regularization creates sparse model

```
In [101]: layer = keras.layers.Dense(100, activation="elu",
                                     kernel_initializer="he_normal",
                                     kernel_regularizer=keras.regularizers.l2(0.01))
          # or l1(0.1) for ℓ1 regularization with a factor or 0.1
          # or l1_l2(0.1, 0.01) for both ℓ1 and ℓ2 regularization, with factors 0.1 and 0.01 respectively
```

Dropout Regularization

- Dropout regularization
  - Most popular regularization technique for deep neural networks
  - 1-2% accuracy boost even for state-of-the-art neural networks
- Dropout algorithm
  - At every training step, every neuron has probability p of being dropped out
    - Dropped out neuron is temporarily ignored during training step
    - May become active again in next training step
  - After training, neurons are no longer dropped out
  - Typical dropout rates: 10-50%



Dropout Regularization

- Dropout in Keras

In [104]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accuracy"])
n_epochs = 2
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid))
```

Monte Carlo Dropout Regularization

- Improvement to prediction of trained model
  - Make multiple predictions while dropout is active
  - Average over all predictions
- Results in more reliable prediction

```
In [111]: y_probas = np.stack([model(X_test_scaled, training=True)
                               for sample in range(100)])
          y_proba = y_probas.mean(axis=0)
```

Max-Norm Regularization

- Max-norm regularization constrains weight of incoming connections
  - For each neuron: $\| \boldsymbol{w} \|_2 \leq r$
  - If limit is exceeded, rescale weights after each training step: $\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\| \mathbf{w} \|_2}$

- In Keras, set kernel_constraint argument:

```
In [125]: layer = keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal",
                                     kernel_constraint=keras.constraints.max_norm(1.))
```

Practical Guidelines

- Recommended default configuration
- Should not require much hyperparameter tuning

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Practical Guidelines

- If network is stack of dense layers, then it can self-normalize
  - Standardize input features

Hyperparameter	Default value
Kernel initializer:	LeCun initialization
Activation function:	SELU
Normalization:	None (self-normalization)
Regularization:	Early stopping
Optimizer:	Nadam
Learning rate schedule:	Performance scheduling

Practical Guidelines

- If you need sparse model
  - Use ℓ_1 regularization (and zero tiny weights after training)
- If you need low-latency model
  - Use fewer layers
  - Fold Batch Normalization into previous layer
  - Use fast activation function, such as leaky ReLU or ReLU
  - Use sparse model
  - Reduce float precision to 32, 16, or 8 bits
- If building risk-sensitive application
  - Use MC Dropout to get more reliable probability estimates
- TensorFlow Model Optimization Toolkit can help

Summary

- Vanishing/exploding gradients problems
- Reusing pretrained layers
- Faster Optimizers
- Avoiding overfitting through regularization

UMassAmherst

The Commonwealth's Flagship Campus