CS5489 - Machine Learning

Lecture 10a - Deep Learning

Prof. Antoni B. Chan

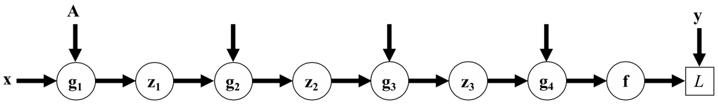
Dept. of Computer Science, City University of Hong Kong

Outline

- Going deeper
  - ReLU and Batchnorm
- Optimization methods
- Deep architectures and Image classification
- Transfer learning

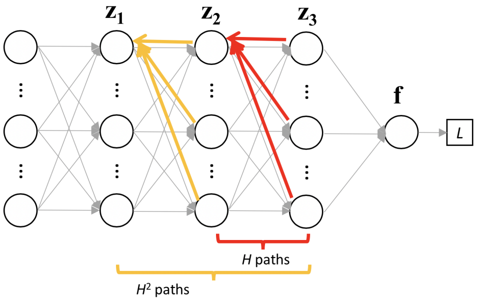
Problems with Going Deeper

- **Vanishing Gradient Problem 1**
  - successive multiplications of small gradients gives smaller gradients, and converges to 0
  - the gradients backpropagated to the first few layers has small signal.
- Example: for 4-layers,



$$\frac{dL}{d\mathbf{a}_j} = \frac{d\mathbf{g}_1^T}{d\mathbf{a}_j}\frac{d\mathbf{z_1}^T}{d\mathbf{g}_1}\frac{d\mathbf{g}_2^T}{d\mathbf{z_1}}\frac{d\mathbf{z_2}^T}{d\mathbf{g}_2}\frac{d\mathbf{g}_3^T}{d\mathbf{z_2}}\frac{d\mathbf{z_3}^T}{d\mathbf{g}_3}\frac{d\mathbf{g}_4^T}{d\mathbf{z_3}}\frac{d\mathbf{f}^T}{d\mathbf{g}_4}\frac{dL}{d\mathbf{f}}$$

- **Vanishing Gradient Problem 2**
  - using backprop, the gradient at a node is the summation over $O(H^D)$ paths
    - D is the number of layers to the output layer.
    - H is the number of nodes in the layer.
  - the original loss signal gets "washed out".



- **Dataset Size**
  - a "small" network with just 40 inputs, 30 hidden nodes, and 1 output has ~1200 parameters.
  - if we don't have enough samples:
    - large variance in the parameter estimator (what you get may be far from the truth)
    - deeper networks are more complex, which are easier to overfit the training data.

- *How many samples do we need?*
  - Theorem (Bartlett, Maiorov, Meir, 1998)

> Suppose $\mathcal{N}$ is a feed-forward network with W weights, L layers, and all non-output gates having a fixed piecewise-polynomial activation function with a fixed number of pieces (e.g., ReLU). Then $VCdim(\mathcal{N}) = O(WL\log W + WL^2)$.

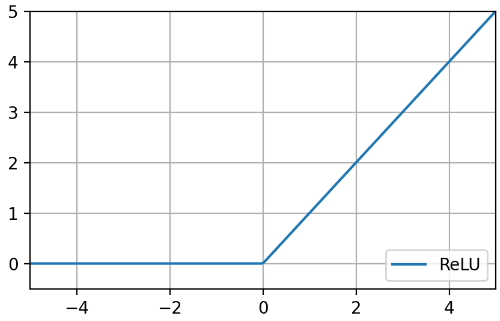
- If the sample size is large compared to the VC dimension, then the learned classifier will generalize well.
  - for the same number of parameters, the deeper network requires more data (WL^2).
  - increasing the number of weights, requires a super-linear increase in sample size ($W\log W$).

ReLU activation function

- Rectified Linear Unit: $\mathrm{ReLU}(z) = \max(0, z)$
  - easier to train with: gradient is either 0 or 1.
  - faster: don't need to calculate exponential
  - sparse representation: most nodes will output zero.

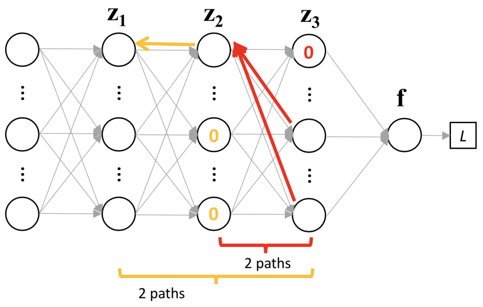
```
In [10]: actfig
```

Out[10]:



Advantage of Sparsity

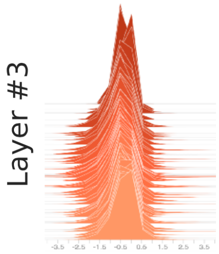
- if a hidden node $h = 0$, then $\frac{dL}{dh} = 0$.
  - This blocks some paths when computing the gradients.
  - Gradient signal is less washed out.
  - Reduces the vanishing gradient problem.



Better Network Parameterization

- There are equivalent parameterizations of the network by scaling up/down the weights in successive layers.
  - $f(\mathbf{x}) = \mathbf{A}^T r(\mathbf{B}^T\mathbf{x}) = \frac{1}{\epsilon}\mathbf{A}^T r(\epsilon\mathbf{B}^T\mathbf{x})$
    - $\epsilon > 0$ and $r(\cdot)$ is the ReLU activation.
- **Problem:**

- "internal covariate shift" - change in the distribution of activations during training, due to changes in the parameters.



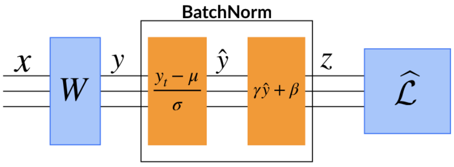
Why is it bad?

- suppose, we have a linear network:
- $y = xw_1w_2w_3w_4$
- gradient of each layer is $(g_1, \cdots, g_4)$
- update the parameters with GD:
  - $y = x(w_1 - \eta g_1)(w_2 - \eta g_2)(w_3 - \eta g_3)(w_4 - \eta g_4)$
    - there are many higher-order terms, e.g., $w_1 w_2 \eta^2 g_3 g_4$
  - although w_i are updated independently, they strongly affect each other.
  - hence, if the distribution of activations changes in 1 layer, then all layers are affected, and we need to adjust other layers.

Solution: Batch Normalization

- For each node in each layer, normalize the outputs to zero mean and unit variance, *over each mini-batch*.
  - this is analogous to the idea of normalizing the input feature vector to (0,1) Gaussian with standard ML models!

- Place batchnorm layer after linear transformation.



- Let $\{y_i\}_{i=1}^N$ be the output of the linear transform in one minibatch.
- For each node (dimension) in the layer:
  - normalize: $\hat{y}_i = \frac{y_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$
    - μ, σ^2 are the mean and variance of $\{y_i\}$ in the mini-batch.
    - ϵ is a small constant for numerical stability.
  - scale-shift: $z_i = \gamma \hat{y}_i + \beta$
    - γ, β are learnable parameters
    - puts the output in the proper regime of the non-linear activation.
  - The final distribution has mean β and variance γ^2.

- Notes:
  - batchnorm is applied to each node independently.
  - should put the batchnorm layer after the linear transformation layer.
  - the bias of the linear layer is not necessary since it is removed by batchnorm

- Training:
    - gradients can be computed through the batchnorm layer as usual.
- Training effects:
    - training is accelerated; can use higher learning rates
    - more stable gradients during training
        - increasing the scale of the activations decreases the gradient
        - self-correcting stabilization.
    - better generalization
        - no need for dropout or L2 regularization.

Example: MNIST

- for each Conv2D/Dense layer:
    - change activation to linear (default); remove bias term
    - append batch-norm and ReLU activation

In [11]:

```
def build_nn():
    K.clear_session()  # cleanup
    random.seed(4487); tf.random.set_seed(4487) # initialize random seed

    # build the network
    nn = Sequential()
    nn.add(Conv2D(10, (5,5), strides=(2,2), input_shape=(28,28,1),
                  padding='same', use_bias=False))
    nn.add(BatchNormalization(axis=3)) # apply batchnorm on channels
    nn.add(Activation("relu"))
    nn.add(Conv2D(40, (5,5), strides=(2,2), padding='same', use_bias=False))
    nn.add(BatchNormalization(axis=3))
    nn.add(Activation("relu"))
    nn.add(Conv2D(80, (5,5), strides=(1,1), padding='same', use_bias=False))
    nn.add(BatchNormalization(axis=3))
    nn.add(Activation("relu"))
    nn.add(Flatten())
    nn.add(Dense(units=50, use_bias=False))
    nn.add(BatchNormalization())
    nn.add(Activation("relu"))
    nn.add(Dense(units=10, activation='softmax'))

    return nn
```

In [12]:

```
nn = build_nn()

# setup early stopping callback function
earlystop = keras.callbacks.EarlyStopping(
    monitor='val_loss',      # look at the validation loss
    min_delta=0.0001,        # threshold to consider as no change
    patience=5,              # stop if 5 epochs with no change
    verbose=1, mode='auto'
)
callbacks_list = [earlystop]

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.SGD(learning_rate=0.02, momentum=0.9, nesterov=True),
          metrics=['accuracy'])
history = nn.fit(vtrainI, vtrainYb, epochs=100, batch_size=50,
                 callbacks=callbacks_list,
                 validation_data=validsetI, verbose=False)
```

```
Metal device set to: Apple M1 Max
```

```
2023-01-23 15:43:30.688074: I tensorflow/core/common_runtime/pluggable_device/plug
gable_device_factory.cc:305] Could not identify NUMA node of platform GPU ID 0, de
faulting to 0. Your kernel may not have been built with NUMA support.
2023-01-23 15:43:30.688564: I tensorflow/core/common_runtime/pluggable_device/plug
```

```
gable_device_factory.cc:271] Created TensorFlow device (/job:localhost/replica:0/t
ask:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevice (device: 0, name:
METAL, pci bus id: <undefined>)
2023-01-23 15:43:31.399733: W tensorflow/core/platform/profile_utils/cpu_utils.cc:
128] Failed to get CPU frequency: 0 Hz
2023-01-23 15:43:31.593293: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
2023-01-23 15:43:36.994830: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

```
Epoch 13: early stopping
```

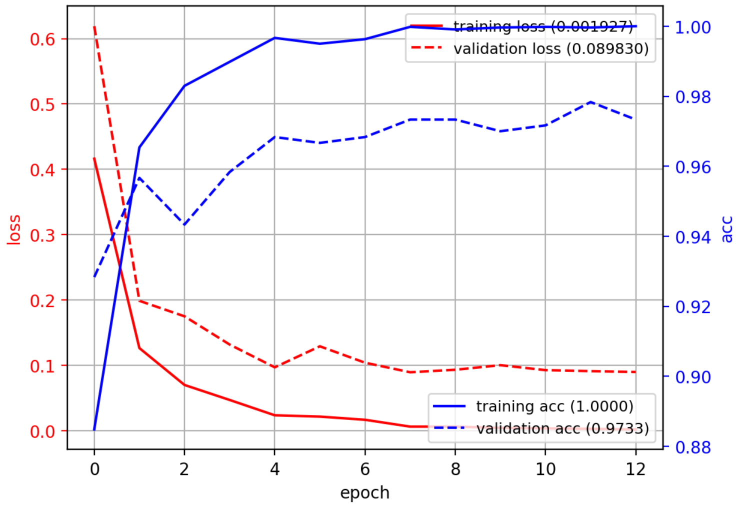
- Test results
    - compared with L2-regularization (0.971), dropout (0.972), ensemble (0.982)

In [13]:

```
plot_history(history)

predY = argmax(nn.predict(testI, verbose=False), axis=-1)
acc = metrics.accuracy_score(testY, predY)
print("test accuracy:", acc)
```

```
2023-01-23 15:44:02.983904: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

```
test accuracy: 0.9817
```



Outline

- Going deeper
    - ReLU and Batchnorm
- **Optimization methods**
- Deep architectures and Image classification
- Transfer learning

Optimization with SGD

- Ideally, we would like to use all the samples to compute the gradient, but this is too time consuming.
- Use a *minibatch* (a few samples) at a time to estimate the gradient.
    - creates an unbiased estimator of the gradient.
    - the variance (expected squared error) depends on the number of samples.
        - i.e., the estimated gradient is *noisy*.

Learning rates

- **Problem:** For gradient descent, at the minimum we should have $\frac{dL}{d\mathbf{w}} = 0$.
- *What about for SGD?*
  - at the minimum $\frac{dL}{d\mathbf{w}} \neq 0$ because of noise in the gradient.
    - SGD still moves around.

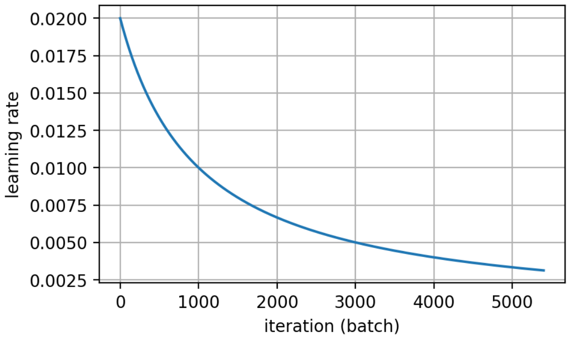
- Solution: reduce the learning rate during the epochs.
  - Examples: for iteration/epoch k,
    - **linear change:** $\eta_k = (1-\alpha)\eta_0 + \alpha\eta_T$, where $\alpha = k/T$, and η_0, η_T given.
    - **decay:** $\eta_k = \frac{1}{1+\delta k}\eta_0$, where $0 < \delta < 1$.
  - we want a small learning rate when we are close to the minimum.
    - needs to be set empirically by examining the learning curves.

Example: Keras decay

- use the built-in `decay` parameter.
  - applied after each batch.

In [14]:

```
plt.figure(figsize=(5,3))
its = arange(0,50*5400/50)  # 50 epochs, 5400/50 iterations per epoch
lr = 0.02*(1./(1+its*1e-3))
plt.plot(its, lr)
plt.grid(); plt.xlabel('iteration (batch)'); plt.ylabel('learning rate');
```



In [15]:

```
nn = build_nn()

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.SGD(learning_rate=0.02,
                                          decay=1e-5,  # decay LR each iteration (batch)
                                          momentum=0.9, nesterov=True),
          metrics=['accuracy'])
history = nn.fit(vtrainI, vtrainYb, epochs=50, batch_size=50,
                 validation_data=validsetI, verbose=False)
```

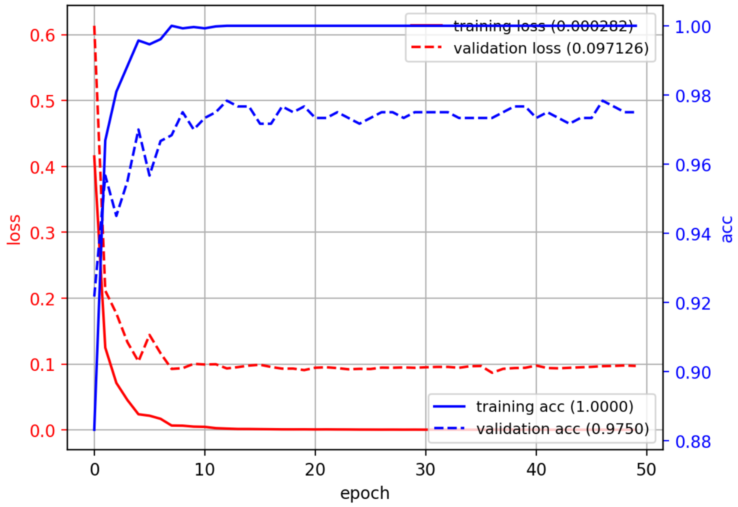
```
2023-01-23 15:44:28.707974: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
2023-01-23 15:44:29.851928: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

In [16]:

```
plot_history(history)

predY = argmax(nn.predict(testI, verbose=False), axis=-1)
acc = metrics.accuracy_score(testY, predY)
print("test accuracy:", acc)
```

```
2023-01-23 15:45:15.109995: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

```
test accuracy: 0.9806
```



Adaptive schedule

- reduce the learning rate when the validation loss no longer improves
    - similar to early stopping criteria
- implemented as a callback function

In [17]:
```
# reduce LR by a factor of 0.1, if no change in 5 epochs
lrschedule = keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
                                 factor=0.1, patience=5, verbose=1)
callbacks_list = [lrschedule]
```

In [18]:
```
nn = build_nn()

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.SGD(learning_rate=0.02, momentum=0.9, nesterov=True),
          metrics=['accuracy'])

history = nn.fit(vtrainI, vtrainYb, epochs=50, batch_size=50,
                callbacks=callbacks_list,
                validation_data=validsetI, verbose=False)
```

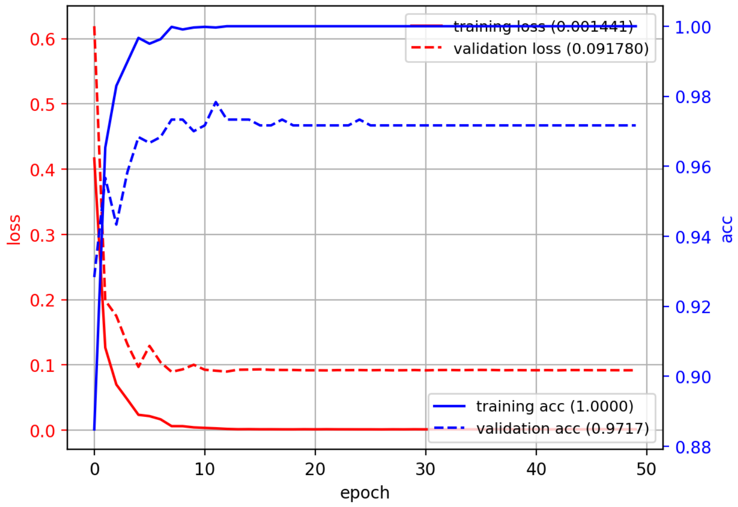
```
2023-01-23 15:45:25.304520: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
2023-01-23 15:45:26.355509: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

```
Epoch 13: ReduceLROnPlateau reducing learning rate to 0.0019999999552965165.

Epoch 18: ReduceLROnPlateau reducing learning rate to 0.0001999999862164259.

Epoch 23: ReduceLROnPlateau reducing learning rate to 1.9999998039565982e-05.

Epoch 28: ReduceLROnPlateau reducing learning rate to 1.99999976757681e-06.

Epoch 33: ReduceLROnPlateau reducing learning rate to 1.99999976757681e-07.

Epoch 38: ReduceLROnPlateau reducing learning rate to 1.9999997391551008e-08.

Epoch 43: ReduceLROnPlateau reducing learning rate to 1.999999810209374e-09.

Epoch 48: ReduceLROnPlateau reducing learning rate to 1.9999997213915323e-10.
```

In [19]:
```
plot_history(history)

predY = argmax(nn.predict(testI, verbose=False), axis=-1)
acc = metrics.accuracy_score(testY, predY)
print("test accuracy:", acc)
```

```
2023-01-23 15:46:10.480098: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

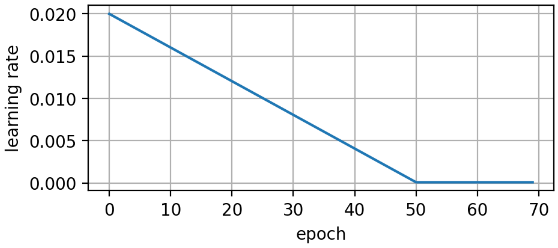
```
test accuracy: 0.9816
```



Fixed schedule

- specify our own schedule using callback `LearningRateScheduler`
- pass a schedule function
- inputs are the epoch and current learning rate.
- outputs the learning rate for this epoch.

In [20]:
```
def sc(epoch, curlr):
    alpha = minimum(epoch/50, 1.)
    return 0.02*(1-alpha)+0.0001*alpha
epoch = arange(0,70)
plt.figure(figsize=(5,2))
plt.plot(epoch, sc(epoch,None))
plt.grid(); plt.xlabel('epoch'); plt.ylabel('learning rate');
```



In [21]:
```
nn = build_nn()

# learning rate schedule
lrschedule = keras.callbacks.LearningRateScheduler(sc, verbose=0)
callbacks_list = [lrschedule]

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.SGD(learning_rate=0.02, momentum=0.9, nesterov=True),
          metrics=['accuracy'])

history = nn.fit(vtrainI, vtrainYb, epochs=50, batch_size=50,
                 callbacks=callbacks_list,
                 validation_data=validsetI, verbose=False)
```

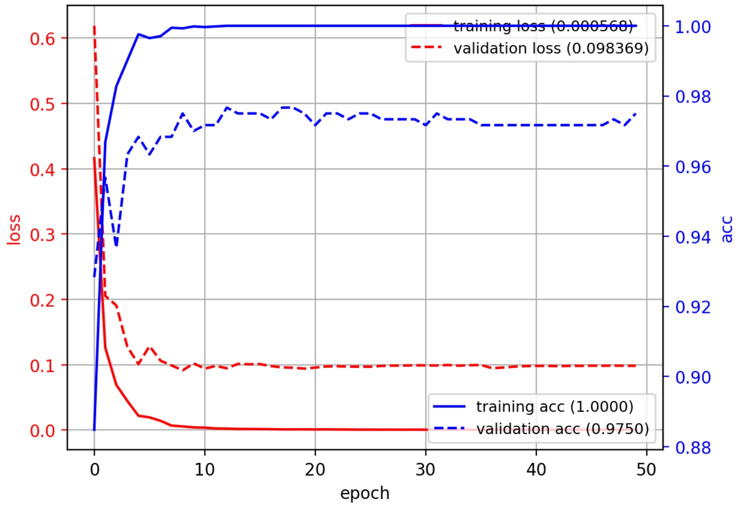
```
2023-01-23 15:46:20.938007: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
2023-01-23 15:46:22.006395: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

In [22]:

```
plot_history(history)

predY = argmax(nn.predict(testI, verbose=False), axis=-1)
acc = metrics.accuracy_score(testY, predY)
print("test accuracy:", acc)
```

```
2023-01-23 15:47:05.917733: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

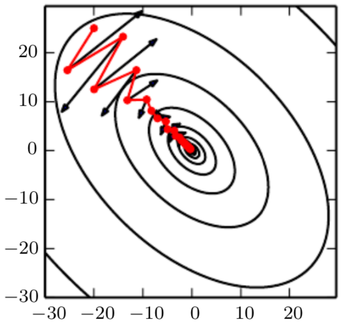
```
test accuracy: 0.9819
```



Momentum

- **Problem**: The estimated gradient is noisy, can jump around.
- **Solution**: keep a running average of the gradients across mini-batches.
  - velocity: $\mathbf{v}^{(t)} = \alpha \mathbf{v}^{(t-1)} - \eta \left.\frac{dL}{d\mathbf{w}}\right\vert_{\mathbf{w}^{(t-1)}}$
  - accumulate the gradients
  - α is the momentum hyperparameter; how much it exponentially decays.
  - parameter update: $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} + \mathbf{v}^{(t)}$

- Example:
  - red path is using momentum
  - black arrows show the gradient directions at each step
  - without momentum, the path would oscillate wildly.



Nesterov Momentum

- Compute the gradient *after* the current velocity is applied.
  - interim update: $\tilde{\mathbf{w}} = \mathbf{w}^{(t-1)} + \alpha\mathbf{v}^{(t-1)}$
  - velocity: $\mathbf{v}^{(t)} = \alpha \mathbf{v}^{(t-1)} - \eta

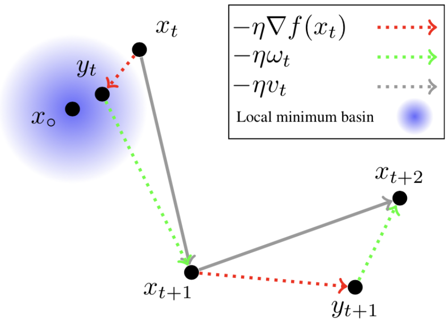
\left.\frac{dL}{d\mathbf{w}}\right\vert_{\tilde{\mathbf{w}}}$

  - parameter update: $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} + \mathbf{v}^{(t)}$
- Adds a *correction factor* to improve convergence (for convex batch case)

Why does SGD work?

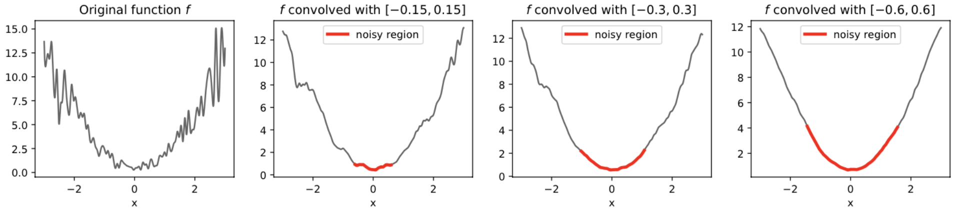
- The loss function has many local minima
- SGD adds "noise" to the true gradient.
  - the noise allows escaping/avoiding/jumping over small local minima.

- Example:
  - red arrow = true gradient
  - green arrow = added noise
  - black arrow = computed gradient



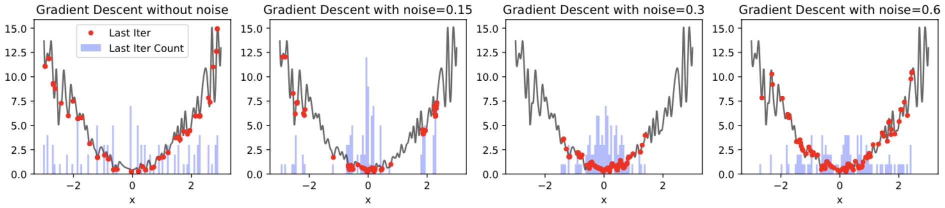
SGD smoothes the loss function

- the added gradient noise is equivalent to convolving the loss function with the noise density.
- higher learning rate $\longrightarrow$ larger noise $\longrightarrow$ smoother loss



- smoother loss removes the local minimum, making it easier to get near the global minimum.

  - but not exactly on it.



- need to reduce the learning rate in stages to converge to the global optimum.

GD, noise=0.3, init=[-3,3]
GD, noise=0.15, init=[-1.5,1.5]
GD, noise=0.05, init=[-0.5,0.5]
GD, noise=0.01, init=[-0.3,0.3]

Optimization with Adaptive Learning Rates

- Introduce a separate learning rate for each parameter, and automatically adapt the learning rates during optimization.
- **AdaGrad** (`keras.optimizers.Adagrad`)
  - adapt individual learning rates by dividing by the square-root of the gradient energy accumulated over the iterations.
    - $\mathbf{g} = \frac{dL}{d\mathbf{w}}$
    - $\mathbf{r}^{(t)} = \mathbf{r}^{(t-1)} + \mathbf{g}^2$
    - $\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)} - \frac{\eta}{\delta + \sqrt{\mathbf{r}}}\mathbf{g}$
  - (operations are element-wise)
- **RMSProp** (`keras.optimizers.RMSprop`)
  - use exponential decay on the accumulated energy:
    - $\mathbf{r}^{(t)} = \rho\mathbf{r}^{(t-1)} + (1-\rho)\mathbf{g}^2$
- **Adam** (`keras.optimizers.Adam`)
  - use momentum with exponential weighting to estimate the gradient and gradient energy.
    - $\mathbf{s}^{(t)} = \rho_1\mathbf{s}^{(t-1)} + (1-\rho_1)\mathbf{g}$
    - $\mathbf{r}^{(t)} = \rho_2\mathbf{r}^{(t-1)} + (1-\rho_2)\mathbf{g}^2$
  - adds a bias correction for these two estimates.
    - $\hat{\mathbf{s}}^{(t)} = \frac{1}{1-\rho_1^t}\mathbf{s}^{(t)}$
    - $\hat{\mathbf{r}}^{(t)} = \frac{1}{1-\rho_2^t}\mathbf{r}^{(t)}$
  - update:
    - $\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)} - \frac{\eta}{\delta + \sqrt{\hat{\mathbf{r}}^{(t)}}}\hat{\mathbf{s}}^{(t)}$

Example

- change the optimizer when compiling the network.

In [23]:

```
 # compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=keras.optimizers.Adam(learning_rate=0.01),   # can set the initial lear
          metrics=['accuracy'])
history = nn.fit(vtrainI, vtrainYb, epochs=100, batch_size=50,
                 callbacks=callbacks_list,
                 validation_data=validsetI, verbose=False)
```

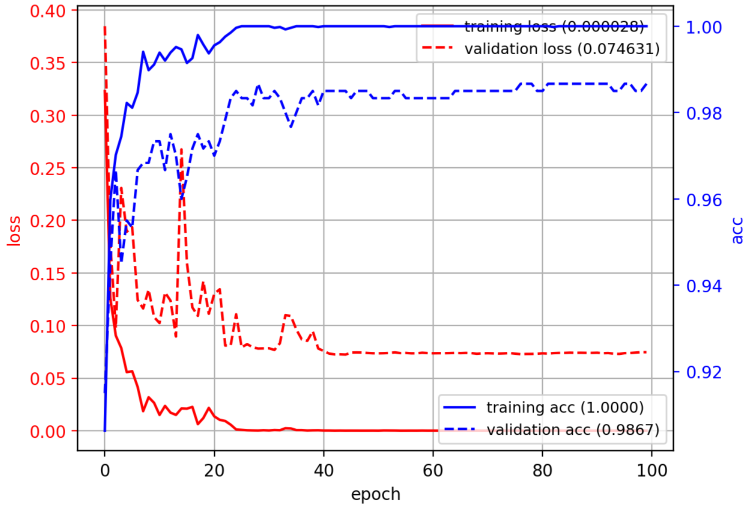
```
2023-01-23 15:50:04.389863: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
2023-01-23 15:50:05.788572: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

In [24]:

```
plot_history(history)

predY = argmax(nn.predict(testI, verbose=False), axis=-1)
acc = metrics.accuracy_score(testY, predY)
print("test accuracy:", acc)
```

```
2023-01-23 15:51:44.202787: I tensorflow/core/grappler/optimizers/custom_graph_opt
imizer_registry.cc:113] Plugin optimizer for device_type GPU is enabled.
```

```
test accuracy: 0.9842
```



Which optimizer is best?

- there's no best optimizer...
- based on the problem and own familiarity with tuning the hyperparmeters.