# Introduction to Deep Learning (I2DL)

## Solution to Exercise 2

# Today

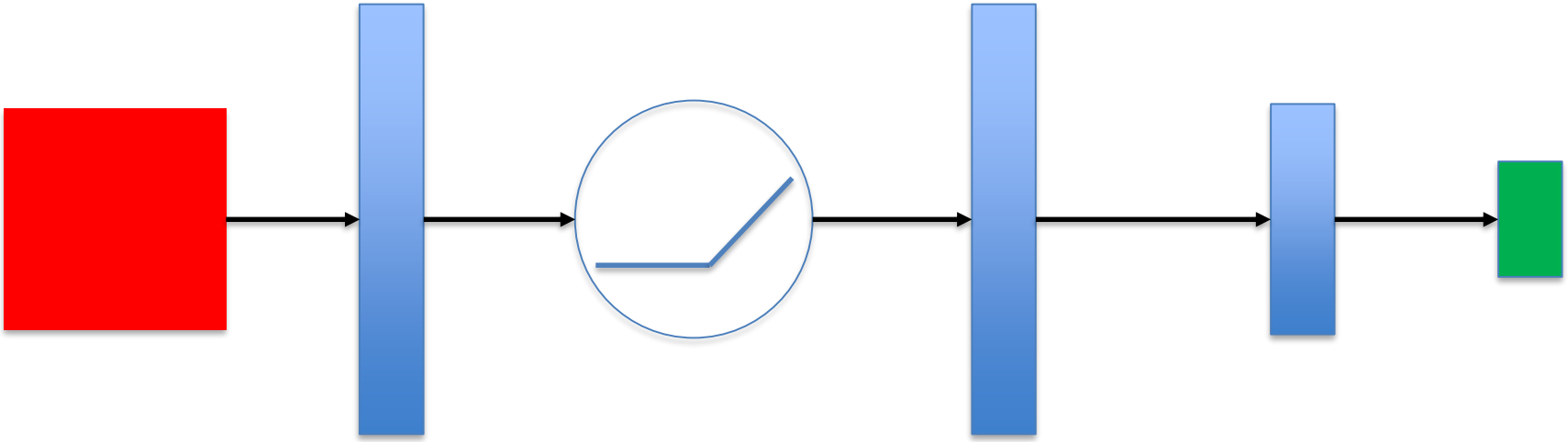
- Discussion of 2nd Exercise Set
  - Your own DL library
    - Affine Layers
    - ReLU
    - L2 Regularisation
    - Optimisers
    - Batch Normalization
    - Dropout

# Goal of Exercise 2

- Build your own DL library with all the modules
- Use you modules to train multi layer network (with Batchnorm and Dropout)

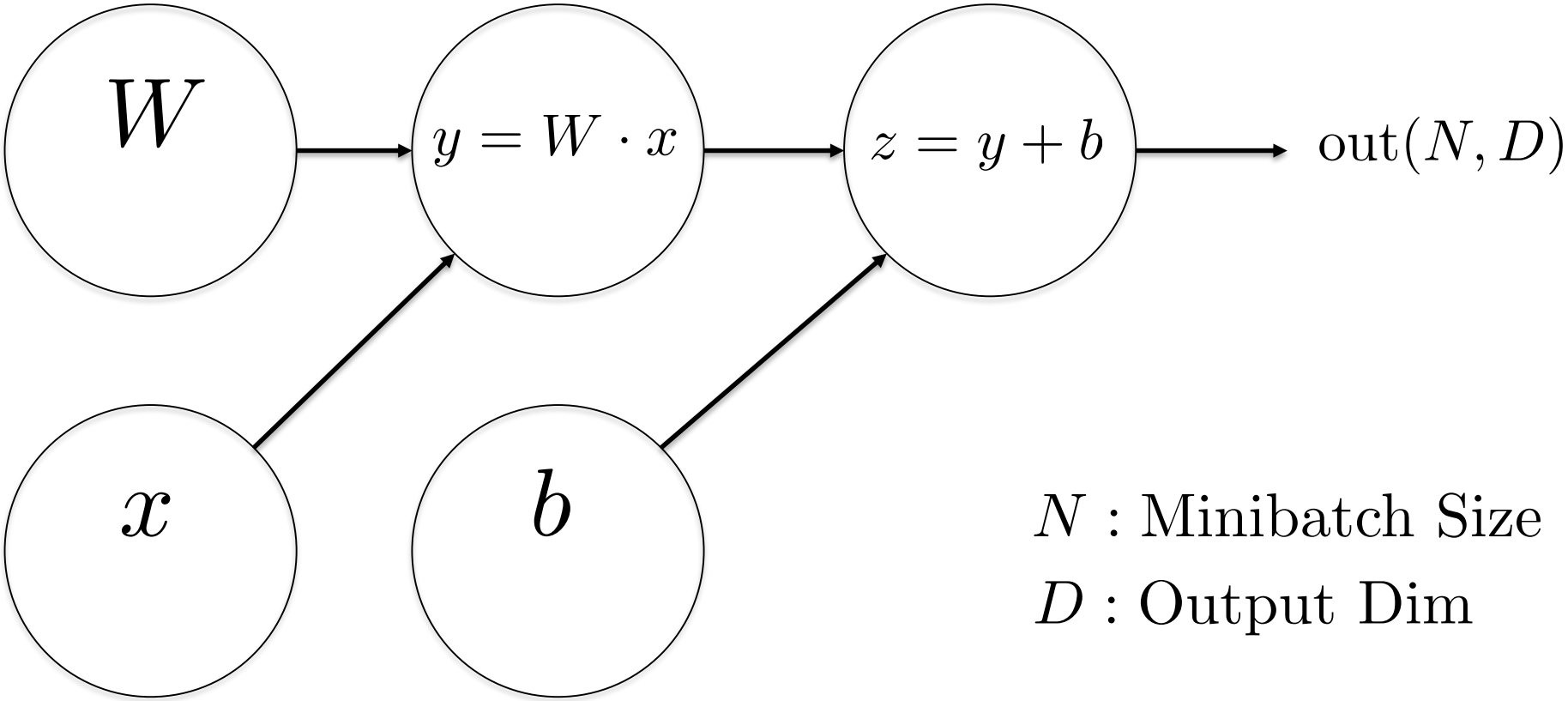
```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

# Two Layer Classifier



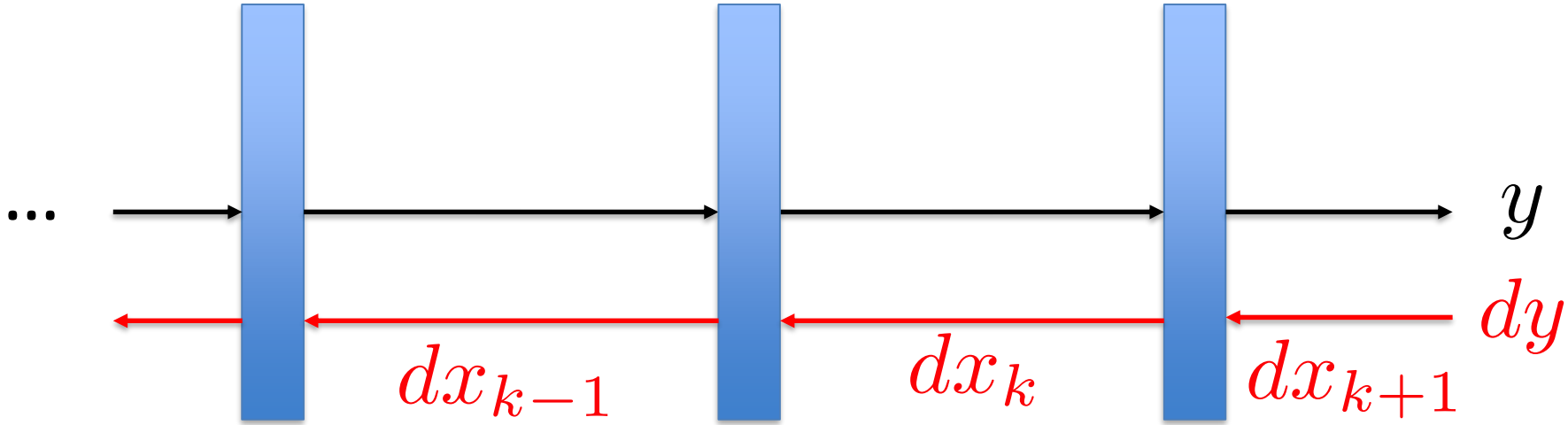
Input Image 1st affine layer ReLU Activation 2nd affine layer Softmax activation Output

# Affine Layer - Forward Pass

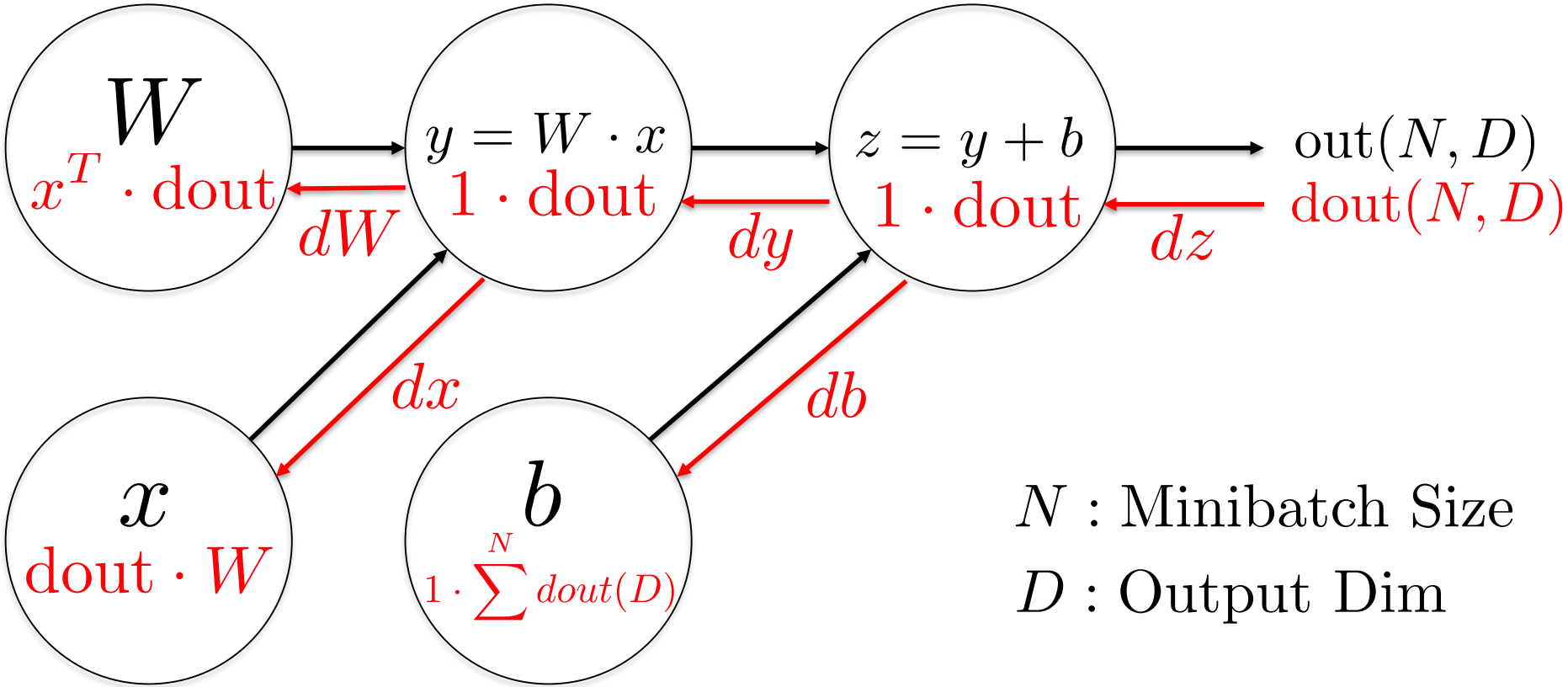


# Backpropagation

$$\mathbf{x}_{k-1} = \mathbf{F}\left(\mathbf{x}_{k-2}\right) \qquad \mathbf{x}_{k} = \mathbf{F}\left(\mathbf{x}_{k-1}\right) \qquad \mathbf{x}_{k+1} = \mathbf{F}\left(\mathbf{x}_{k}\right)$$

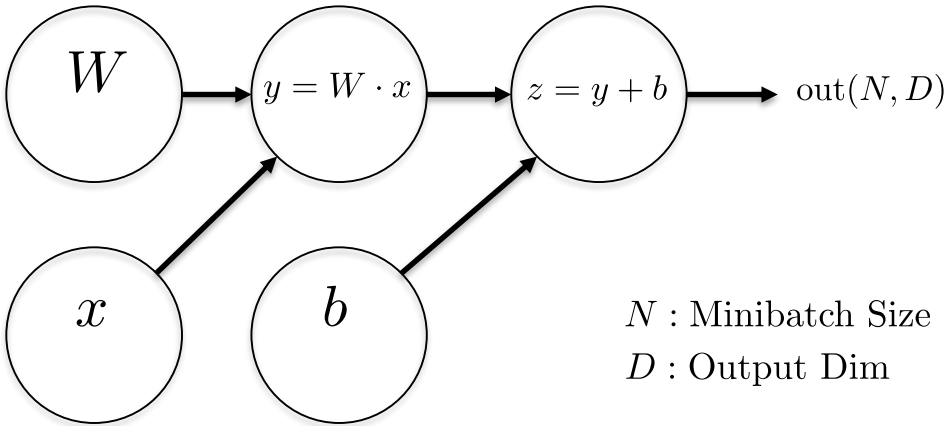


# Affine Layer - Backward Pass

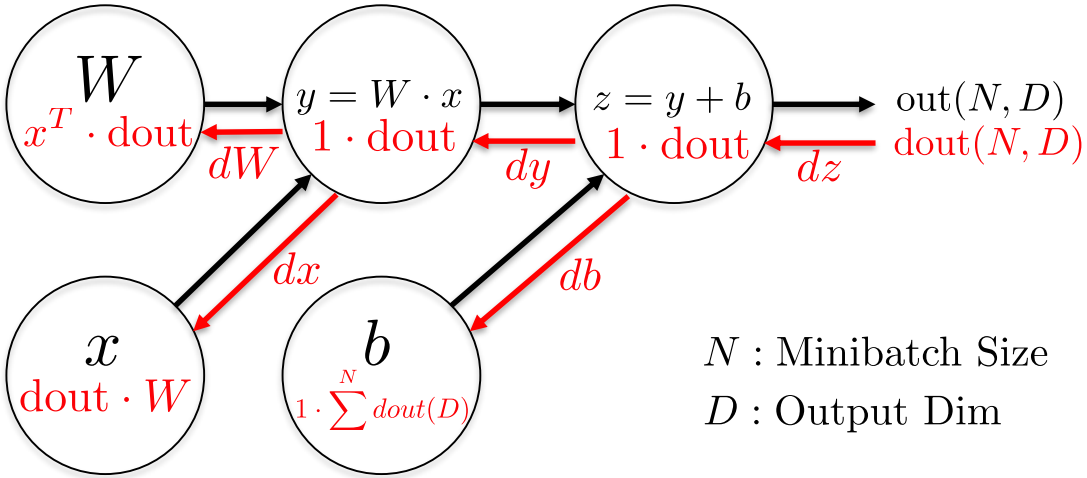


# Affine Layer - Forward Pass

```
#step 0: Prepare input variable
x_reshaped = np.reshape(x, (x.shape[0], -1))

#step1
y = np.dot( x_reshaped, w )

#step2
z = y + b

z = out
```



# Affine Layer - Backward Pass




```
#step2
db = np.sum( dout, axis = 0 , keepdims = True)

#step1
dw = (np.reshape( x, (x.shape[0], -1)).T).dot(dout)
dw = np.reshape(dw, w.shape)
dx = dout.dot(w.T)
dx = np.reshape( dx, x.shape)
```

# ReLU - Forward Pass

$$\text{ReLU}(x) = \begin{cases} x, \text{ if } x > 0 \\ 0, \text{ else} \end{cases}$$

```
################################################################################
# TODO: Implement the ReLU forward pass.                                       #
################################################################################
out = np.maximum( x, 0)
################################################################################
#                             END OF YOUR CODE                                 #
################################################################################
```

# ReLU - Backward Pass

$$\frac{\partial}{\partial x}\text{ReLU}(x) = \begin{cases} 1, \text{ if } x > 0 \\ 0, \text{ else} \end{cases}$$

```
##############################################################################
# TODO: Implement the ReLU backward pass.                                    #
##############################################################################
dx = dout
dx[x < 0] = 0
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```

# Softmax Loss

```python
def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
      class for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    numerical_stabilizer = 1e-14
    probs = np.exp(x - np.max(x, axis=1, keepdims=True))
    probs /= np.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    loss = -np.sum(np.log(probs[np.arange(N), y] + numerical_stabilizer)) / N
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    return loss, dx
```

$$E = -\sum t_i log\left(p_i\right) = -\sum t_i log\left(\frac{e^{y_i}}{\sum_k e^{y_k}}\right)$$

- Softmax Loss = Softmax activation + Cross Entropy Loss
- Derivation of backward pass in appendix

# Sandwich Layers

- Normally not implemented in common DLlibraries
- Useful for more difficult architectures.

```
def affine_relu_forward(x, w, b):
    """
    Convenience layer that perorms an affine transform followed by a ReLU

    Inputs:
    - x: Input to the affine layer
    - w, b: Weights for the affine layer

    Returns a tuple of:
    - out: Output from the ReLU
    - cache: Object to give to the backward pass
    """
    a, fc_cache = affine_forward(x, w, b)
    out, relu_cache = relu_forward(a)
    cache = (fc_cache, relu_cache)
    return out, cache
```

# Solver

```
data = {
  'X_train': # training data
  'y_train': # training labels
  'X_val': # validation data
  'X_train': # validation labels
}
model = MyAwesomeModel(hidden_size=100, reg=10)
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-3,
                },
                lr_decay=0.95,
                num_epochs=10, batch_size=100,
                print_every=100)
solver.train()
```

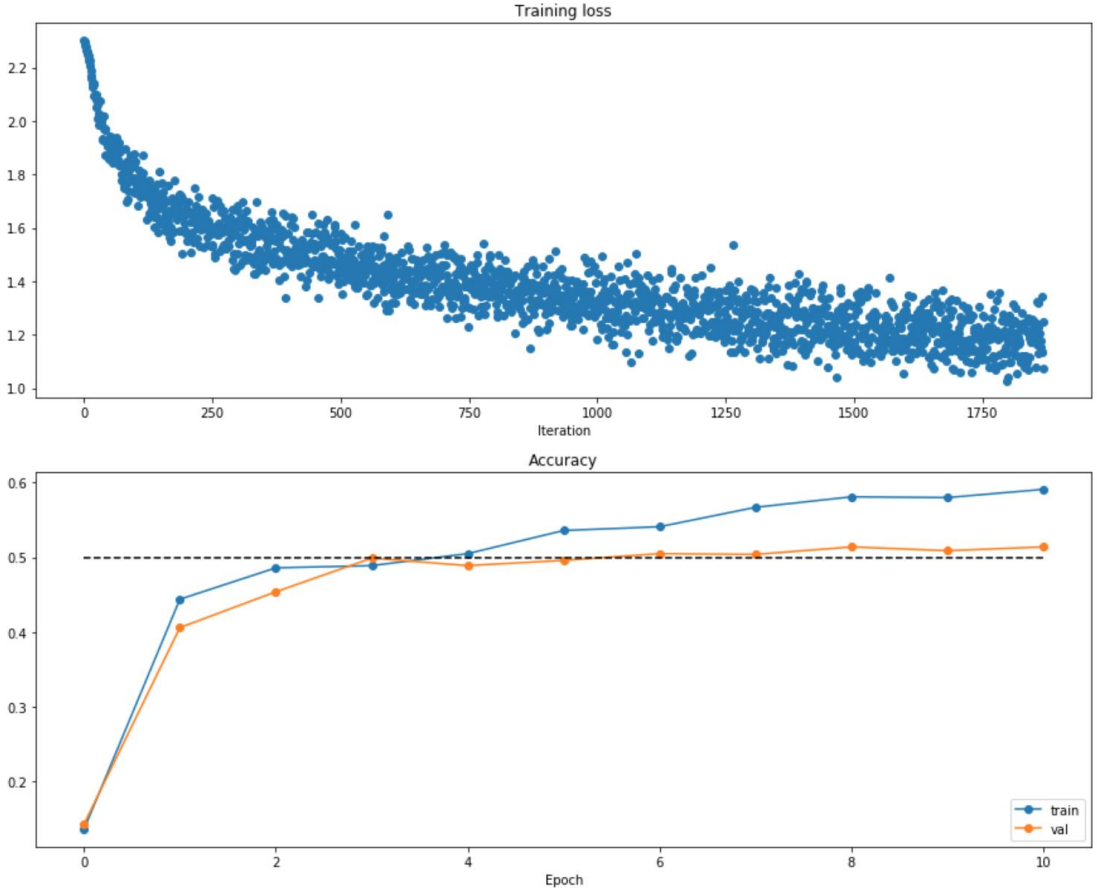
# Solver

- train() function:
  - Iterates through batches until maximal number of iterations/ epochs or convergence/ accuracy (early stopping) is reached
  - step() function for weight update
  - validate() function to check training accuracy
  - checkpoint() function that saves models during training

# Solver: step()

```
def _step(self):
    """
    Make a single gradient update. This is called by train() and should not
    be called manually.
    """
    # Make a minibatch of training data
    num_train = self.X_train.shape[0]
    batch_mask = np.random.choice(num_train, self.batch_size)
    X_batch = self.X_train[batch_mask]
    y_batch = self.y_train[batch_mask]

    # Compute loss and gradient
    loss, grads = self.model.loss(X_batch, y_batch)
    self.loss_history.append(loss)

    # Perform a parameter update
    for p, w in self.model.params.items():
        dw = grads[p]
        config = self.optim_configs[p]
        next_w, next_config = self.update_rule(w, dw, config)
        self.model.params[p] = next_w
        self.optim_configs[p] = next_config
```

# Train/Val


Training loss
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0
250
500
750
1000
1250
1500
1750
Iteration
Accuracy
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10
Epoch
train
val

# Multilayer Neural Network - Init

```
self.params['W0'] = weight_scale * np.random.randn(input_dim, hidden_dims[0])
self.params['b0'] = np.zeros(hidden_dims[0])

for idx in range(len(hidden_dims) - 1):
    weightMatrix = 'W%s' % str(idx + 1)
    biasVector = 'b%s' % str(idx + 1)
    self.params[weightMatrix] = weight_scale * np.random.randn(hidden_dims[idx],
    self.params[biasVector] = np.zeros(hidden_dims[idx + 1])

    for idx in range(len(hidden_dims)):
        if self.use_batchnorm:
            self.params['gamma' + str(idx + 1)] = np.ones(hidden_dims[idx])
            self.params['beta' + str(idx + 1)] = np.zeros(hidden_dims[idx])

lastWeightMatrix = 'W%s' % str(len(hidden_dims))
lastBiasVector = 'b%s' % str(len(hidden_dims))
self.params[lastWeightMatrix] = weight_scale * np.random.randn(hidden_dims[-1],
self.params[lastBiasVector] = np.zeros(num_classes)
```

# Multilayer Neural Network - Forward

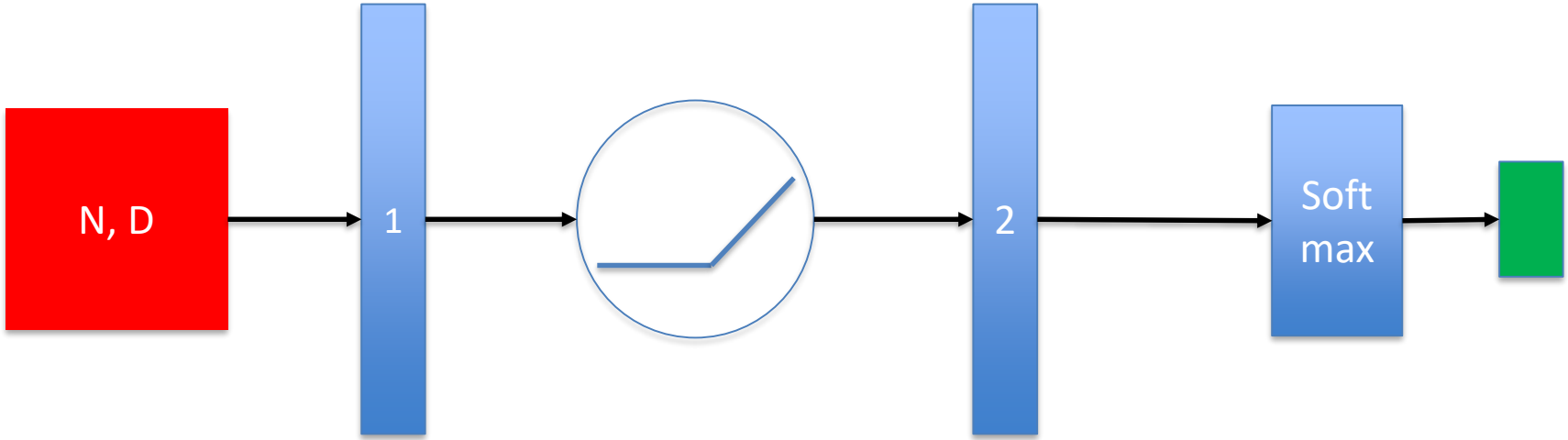
```
out = X
cacheDict = {}
for layer in range(self.num_layers - 1):
    W_str = 'W%s' % layer
    b_str = 'b%s' % layer
    W = self.params[W_str]
    b = self.params[b_str]
    out, cache_affine = affine_forward(out, W, b)

    if self.use_batchnorm:
        out, cache_batch = batchnorm_forward(
            out, self.params['gamma' + str(layer + 1)],
            self.params['beta' + str(layer + 1)],
            self.bn_params[layer])
        cacheDict[(layer, 'batch')] = cache_batch

    out_relu, cache_relu = relu_forward(out)
    cacheDict[(layer, 'affine')] = cache_affine
    cacheDict[(layer, 'relu')] = cache_relu

    if self.use_dropout:
        out, cache_dropout = dropout_forward(out_relu,
                                             self.dropout_param)
        cacheDict[(layer, 'dropout')] = cache_dropout
    else:
        out = out_relu

W_last = self.params['W%s' % str(self.num_layers - 1)]
b_last = self.params['b%s' % str(self.num_layers - 1)]
scores, cache_scores = affine_forward(out, W_last, b_last)
```

# Multilayer Neural Network - Backward

```
# loss
# data effect
loss, dscores = softmax_loss(scores, y)
# regularization effect
for layer in range(self.num_layers):
    W = self.params['W' + str(layer)]
    loss += 0.5 * self.reg * np.sum(W * W)

# gradients
# last layer
dx, dW, db = affine_backward(dscores, cache_scores)
grads['W' + str(self.num_layers - 1)] = dW + self.reg *
    self.params['W' + str(self.num_layers - 1)]
grads['b' + str(self.num_layers - 1)] = db.flatten()

for layer in range(self.num_layers)[::-1][1:]:
    if self.use_dropout:
        dx = dropout_backward(dx, cacheDict[(layer, 'dropout')])

    dx = relu_backward(dx, cacheDict[(layer, 'relu')])

    if self.use_batchnorm:
        dx, dgamma, dbeta = batchnorm_backward(dx,
            cacheDict[(layer, 'batch')])
        grads['gamma' + str(layer + 1)] = dgamma
        grads['beta' + str(layer + 1)] = dbeta

    dx, dW, db = affine_backward(dx, cacheDict[(layer, 'affine')])
    grads['W' + str(layer)] = dW + self.reg *
        self.params['W' + str(layer)]
    grads['b' + str(layer)] = db.flatten()
```

# Multilayer Neural Network

N, D, H1, H2, C = 2, 15, 20, 30, 10
weight scale = 0.05



# Multilayer Neural Network

- Gradient Checking and initial loss

```
Running check with reg =  0
Initial loss:  2.3059736426008888
W0 relative error: 2.80e-07
W1 relative error: 1.03e-06
W2 relative error: 2.43e-07
b0 relative error: 7.87e-09
b1 relative error: 2.62e-08
b2 relative error: 1.53e-10
Running check with reg =  3.14
Initial loss:  6.936351436715421
W0 relative error: 3.39e-08
W1 relative error: 4.73e-08
W2 relative error: 1.88e-08
b0 relative error: 1.13e-08
b1 relative error: 4.41e-09
b2 relative error: 2.17e-10
```

**Regularization = 0:**

-Natural Log 1/C

C: Number of classes

**Regularization = 3.14:**

- Why is additional Regularisation Loss about 4.7?

# Multilayer Neural Network

- Let X be a gaussian M-dimensional vector with zero mean and unit variance. Then

$$E(|X|_2^2) = E(X_1^2 + \cdots + X_M^2) = E(X_1^2) + \ \ldots + E(X_M^2)$$
$$= Var(X_1) + \ \ldots + Var(X_M) = M$$

(as $Var(Y) = E(Y^2) - E(Y)^2 = 1 - 0 = 1$)

- We use a weight scale of 0.05 and since

$$Var(\alpha Y) = \alpha^2 Var(Y)$$

we have

$$E(|0.05 \ \cdot X|_2^2) = 0.05^2 M$$

# Multilayer Neural Network

- As

N, D, H1, H2, C = 2, 15, 20, 30, 10

we get

$$M = \#W_1 + \#W_2 + \#W_3 = (15 \cdot 20) + (20 \cdot 30) + (30 \cdot 10) = 1200$$

and

$$E(|0.05 \cdot X|_2^2) = 3$$

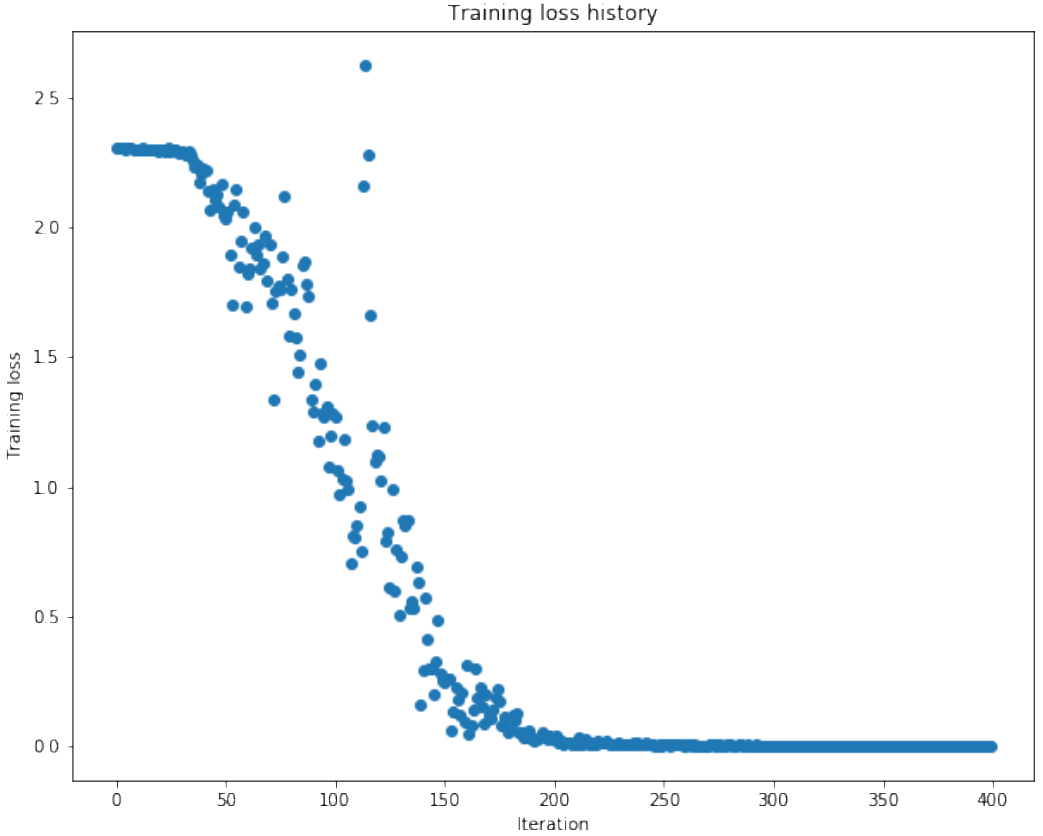
- Together with regularization 3.14 and 0.5 factor

$$Loss_{reg} = 3 * 3.14 * 0.5 = 4{,}71$$

# Overfitting 3-Layer

LR: 1e-2

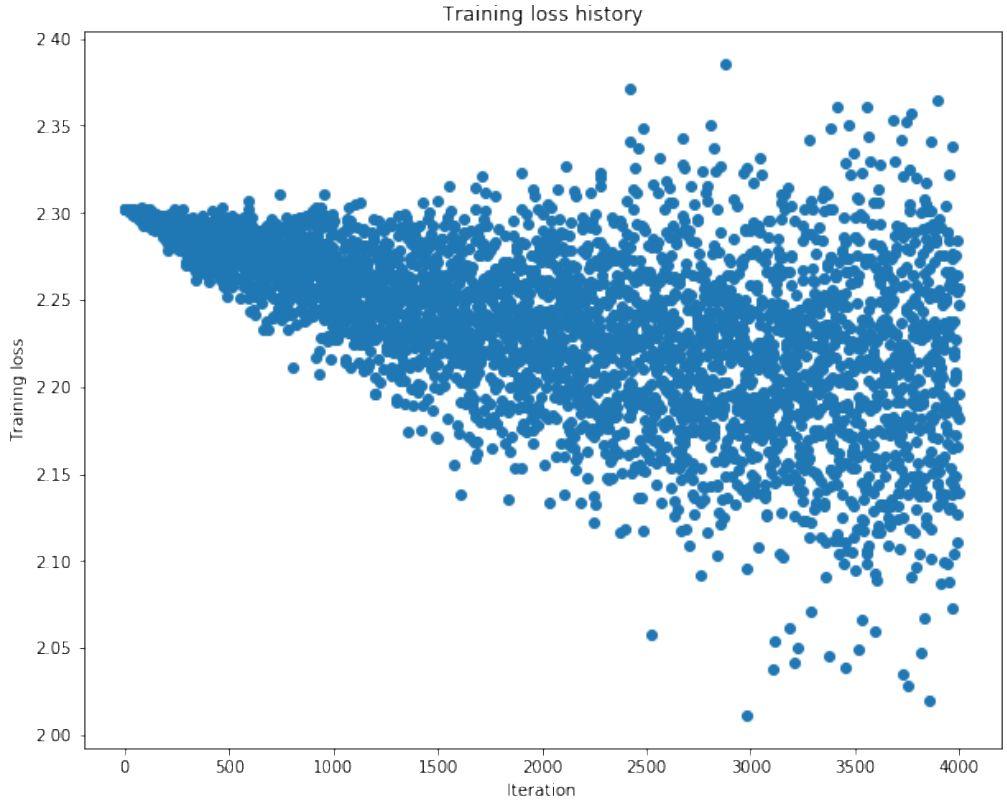
WS: 1e-2



# Overfitting 5-layer

LR: 2e-3
WS: 1e-3

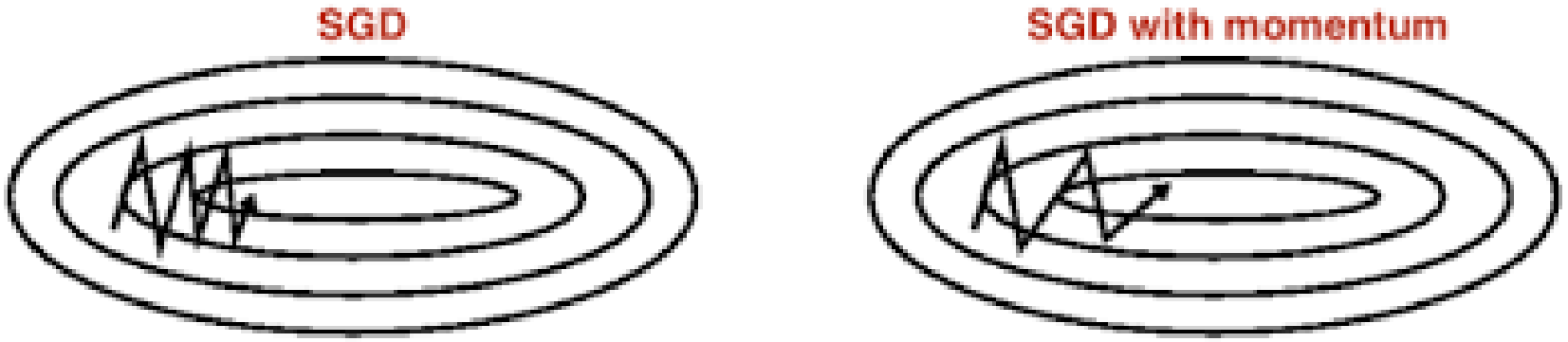
Training a deep neural network is way more difficult and volatile!



# Strategies For Improvement?

- Better initialization methods than Gaussian (lecture)
  - „He-Initalization“ and Relu (default for PyTorch btw): https://arxiv.org/pdf/1502.01852.pdf
  - Glorot/Xavier, etc.
- More effective optimiser
  - SGD + Momentum
  - ADAM (https://arxiv.org/pdf/1412.6980.pdf)
- Batch Normalization
- Dropout

# SGD + Momentum




Taken from the Coursera Course Introduction to Deep Learning (by Higher School of Economics)

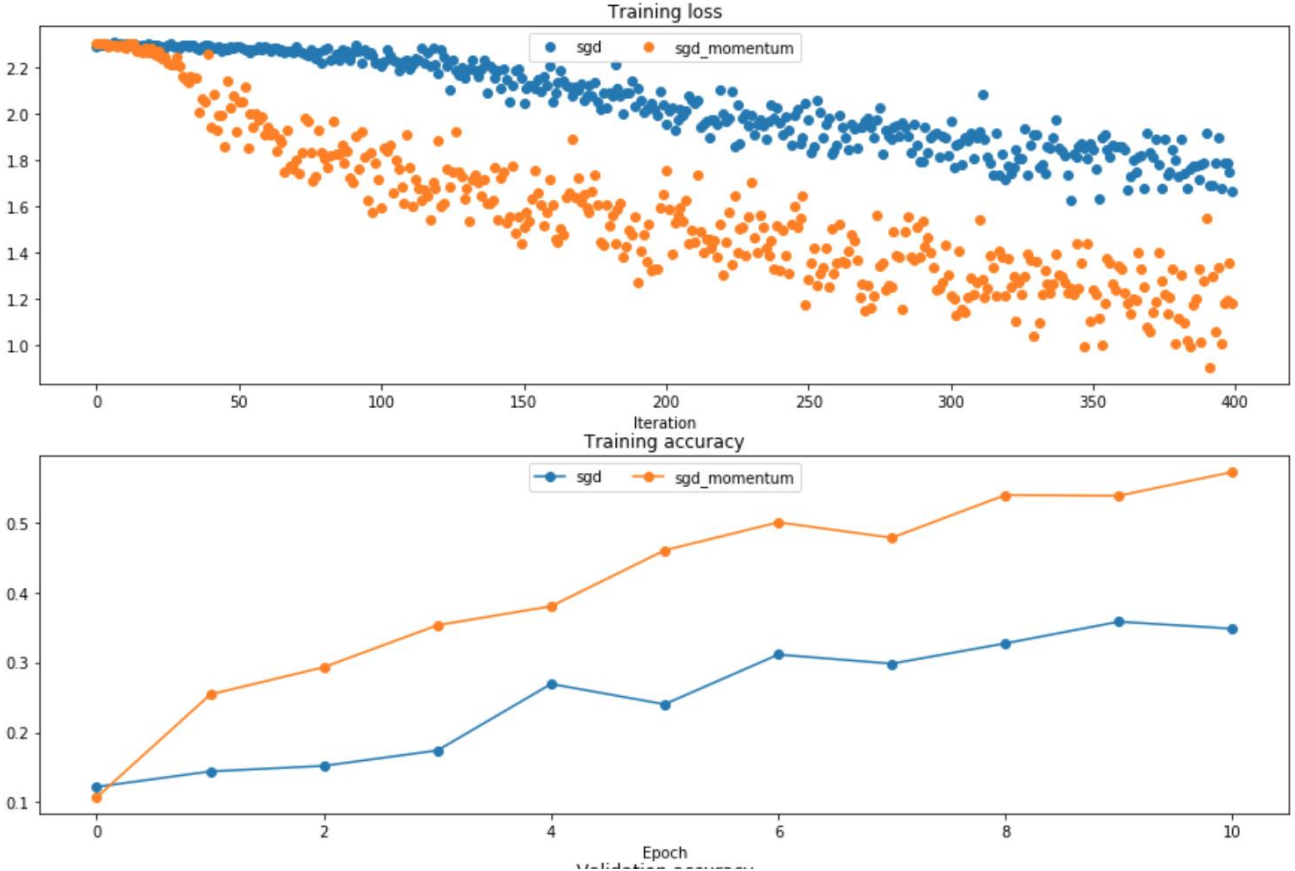
# SGD + Momentum

$$V_t = \mu V_{t-1} - \eta \nabla_w L(W, X, y)$$

$$W = W + V_t$$

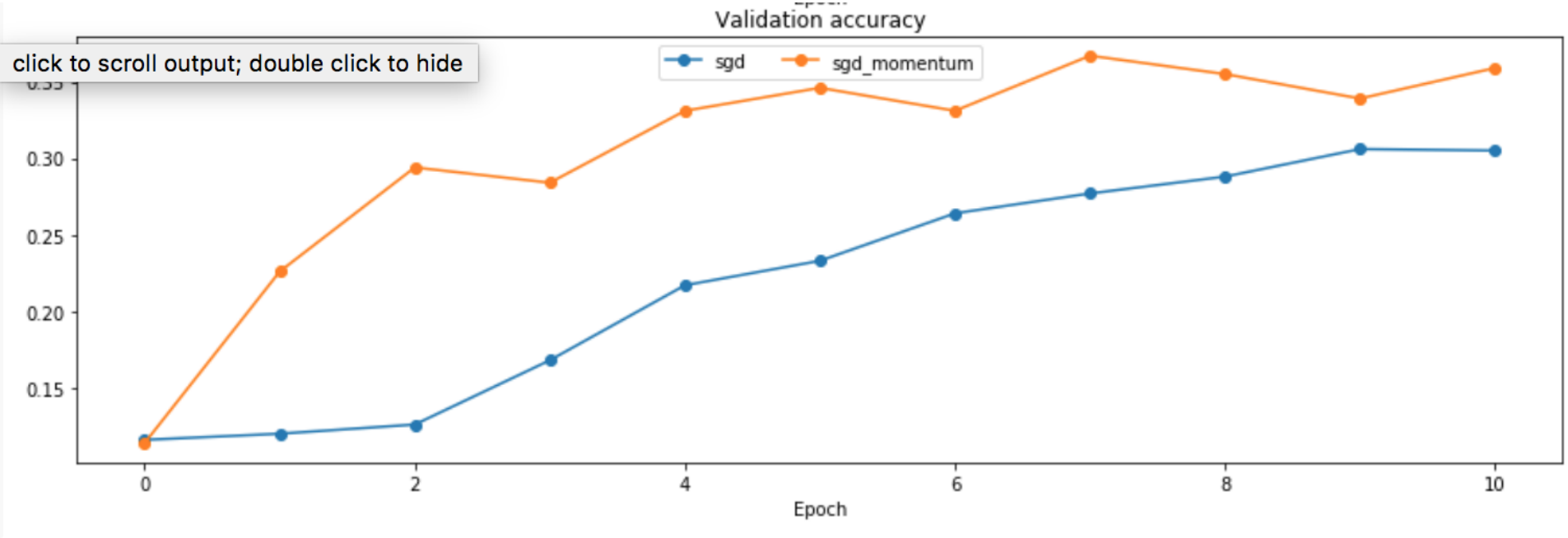
```
next_w = None
##############################################################################
# TODO: Implement the momentum update formula. Store the updated value in    #
# the next_w variable. You should also use and update the velocity v.        #
##############################################################################
mu = config['momentum']
learning_rate = config.get('learning_rate')
v = mu * v - learning_rate * dw
next_w = w + v
config['velocity'] = v
##############################################################################
#                               END OF YOUR CODE                             #
##############################################################################
config['velocity'] = v
```

# SGD + Momentum



# SGD + Momentum

- Validation



# ADAM (Adaptive Movement Estimation)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$ Mean

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$ Variance

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

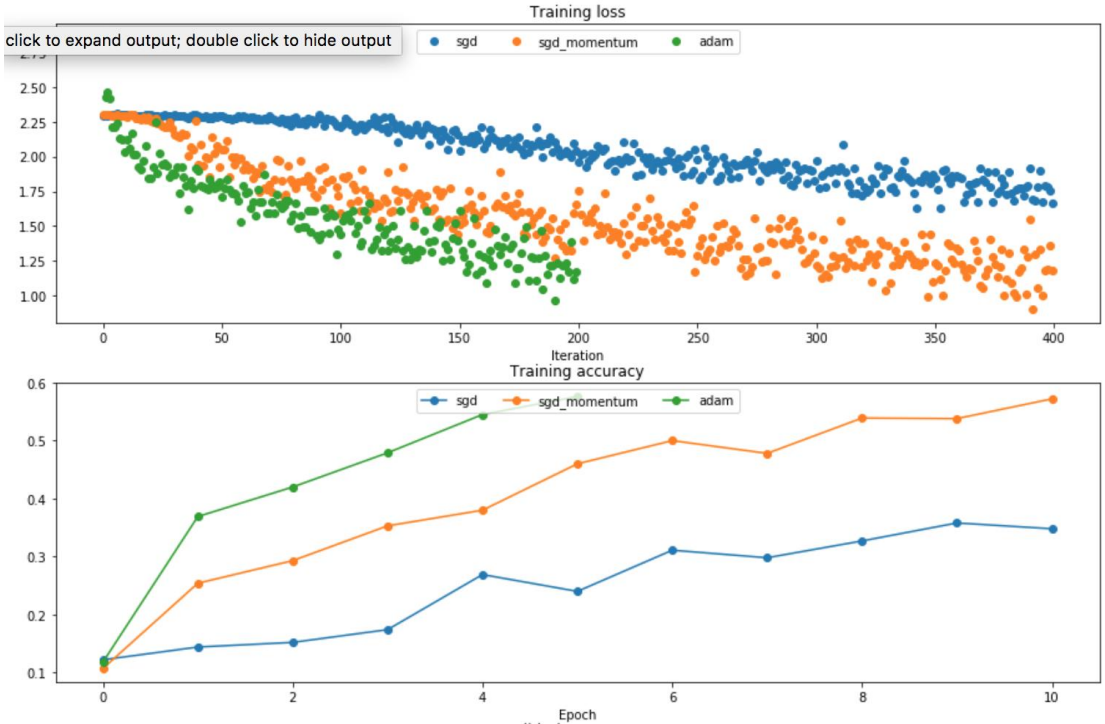
$$W = W - \frac{\eta}{\sqrt{\hat{v}} + \epsilon} \hat{m}_t$$

# ADAM (Adaptive Movement Estimation)

```
m = config['m']
v = config['v']
t = config['t']
beta1 = config['beta1']
beta2 = config['beta2']
learning_rate = config['learning_rate']
eps = config['epsilon']

m = beta1 * m + (1 - beta1) * dx
m_hat = m / (1 - np.power(beta1, t + 1))
v = beta2 * v + (1 - beta2) * (dx ** 2)
v_hat = v / (1 - np.power(beta2, t + 1))
next_x = x - learning_rate * m_hat / (np.sqrt(v_hat) + eps)

config['t'] = t + 1
config['m'] = m
config['v'] = v
```

```
if config is None: config = {}
config.setdefault('learning_rate', 1e-3)
config.setdefault('beta1', 0.9)
config.setdefault('beta2', 0.999)
config.setdefault('epsilon', 1e-8)
config.setdefault('m', np.zeros_like(x))
config.setdefault('v', np.zeros_like(x))
config.setdefault('t', 0)
```

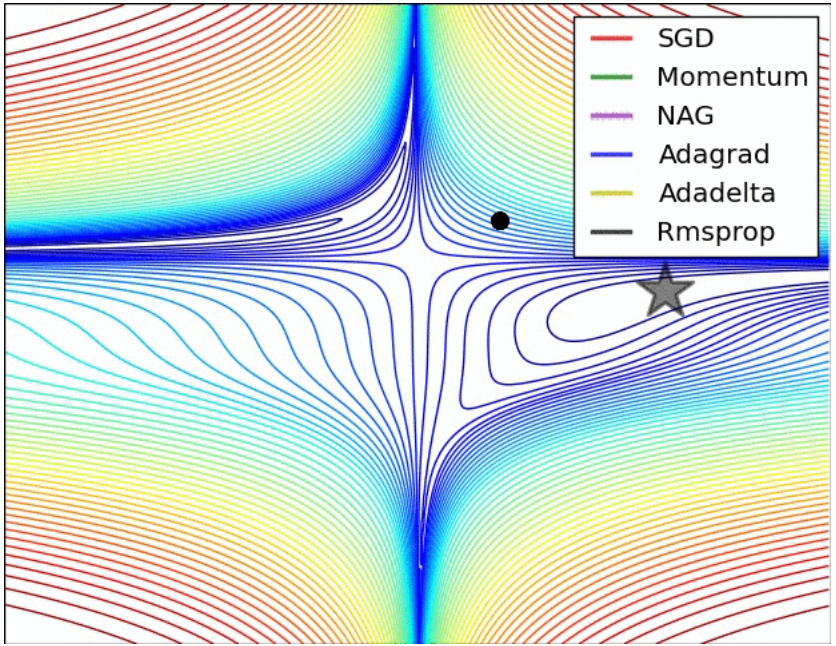
# ADAM
# (Adaptive Movement Estimation)




See (take about 20 minutes and read all of it):
http://ruder.io/optimizing-gradient-descent/index.html#adam

# Which Optimizer to Use?

- The “Oldschool“: SGD + Momentum
- The default: Adam



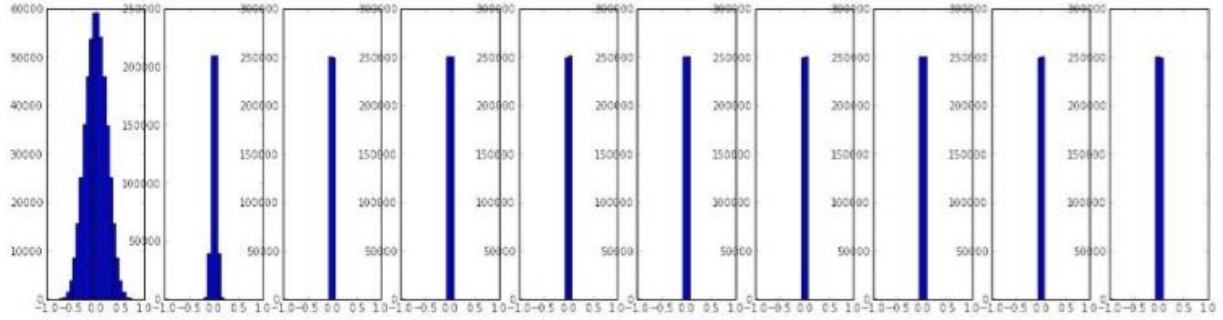
# Strategies for improvement?

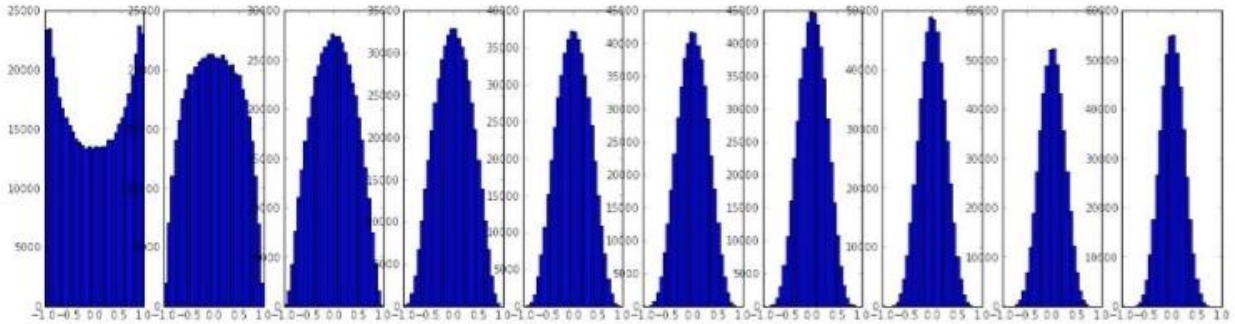
- More effective optimiser
  - SGD + Momentum
  - ADAM (https://arxiv.org/pdf/1412.6980.pdf)
- Batch Normalization
- Dropout

# Recap Lecture:
# Batch Normalization

# Motivation Batchnorm

- All we want is that our activations do not die out





# BatchNorm!

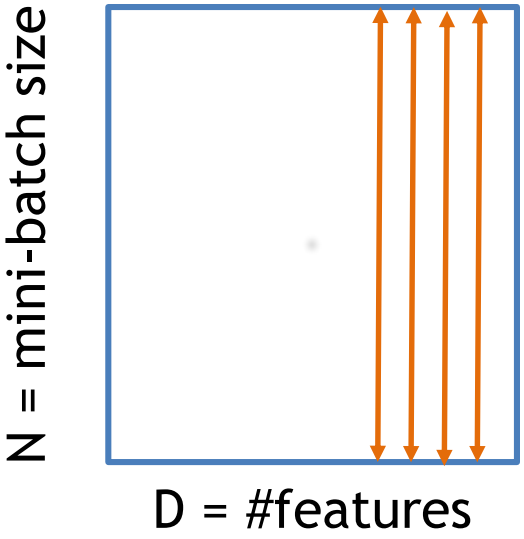
- **Insight:**

  Layers seem to work better with uncorrelated features (zero mean, unit variance) -> see data preprocessing
- **Simple solution and idea of Batch Normalization:**

  Enforce this for every layer by computing mean and std variance and saving the running mean/std var during training to compute it on validation/test

# Batch Normalization

- In each dimension of the features, you have a unit gaussian (in our example)




Mean of your mini-batch examples over feature k

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

# Batch Normalization

- 1. Normalize

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

- 2. Allow the network to change the range

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

backprop

The network *can* learn to undo the normalization

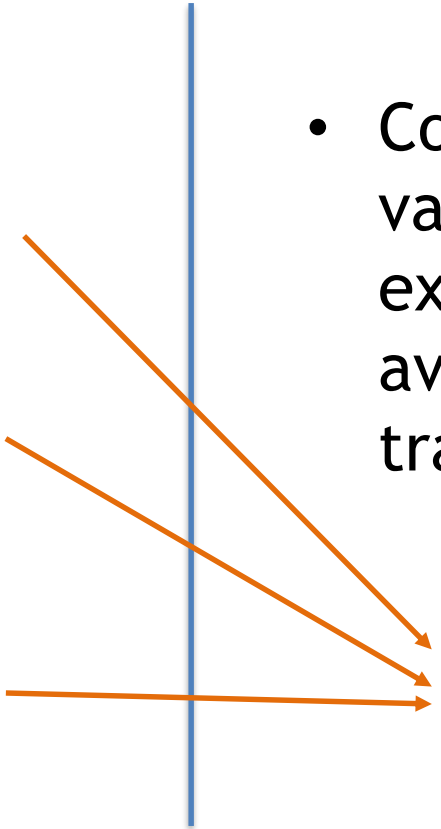
$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

# Batchnorm: Training vs. Test Time

**Training**

- Compute mean and variance from mini-batch 1
- Compute mean and variance from mini-batch 2
- Compute mean and variance from mini-batch 3



**Testing**

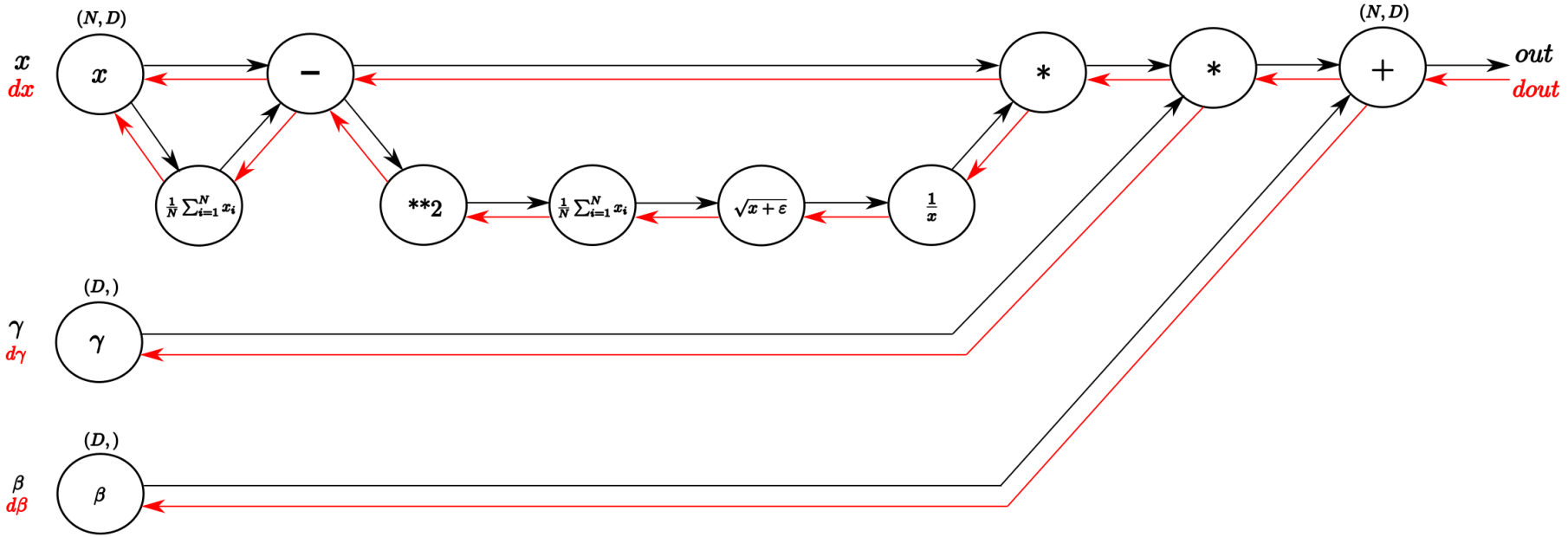
- Compute mean and variance by running an exponentially weighted averaged across training mini-batches

$\mu_{test}$ $\sigma^2_{test}$

# Batchnorm

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$



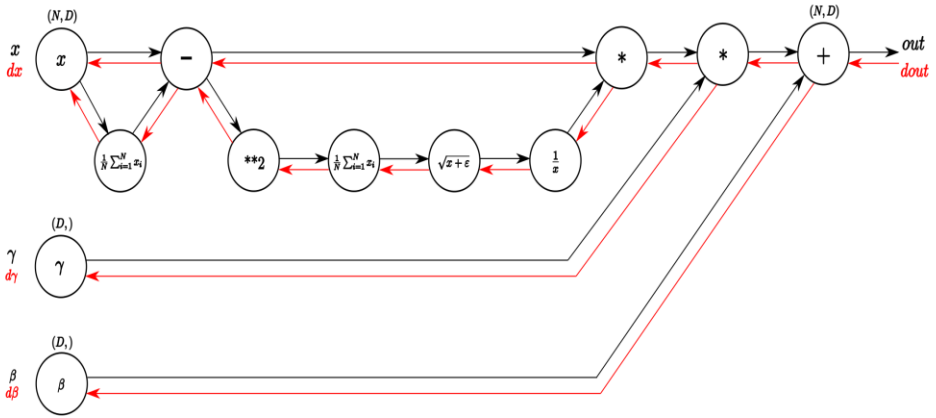

Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Forward Pass

```
#step1: calculate mean
sample_mean = np.mean(x, axis=0)

#step2: subtract mean vector of every trainings example
x_minus_mean = x - sample_mean

#step3: following the lower branch - calculation denominator
sq = x_minus_mean ** 2

#step4: calculate variance
var = 1. / N * np.sum(sq, axis=0)

#step5: add eps for numerical stability, then sqrt
sqrtvar = np.sqrt(var + eps)

#step6: invert sqrtwar
ivar = 1. / sqrtvar

#step7: execute normalization
x_norm = x_minus_mean * ivar

#step8: Combine the two transformation steps
gammax = gamma * x_norm

#step9: Calculate output
out = gammax + beta

running_var = momentum * running_var + (1 - momentum) * var
running_mean = momentum * running_mean + (1 - momentum) * sample_mean

cache = (out, x_norm, beta, gamma, x_minus_mean, ivar, sqrtvar, var, eps)
```

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$




Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Forward Pass

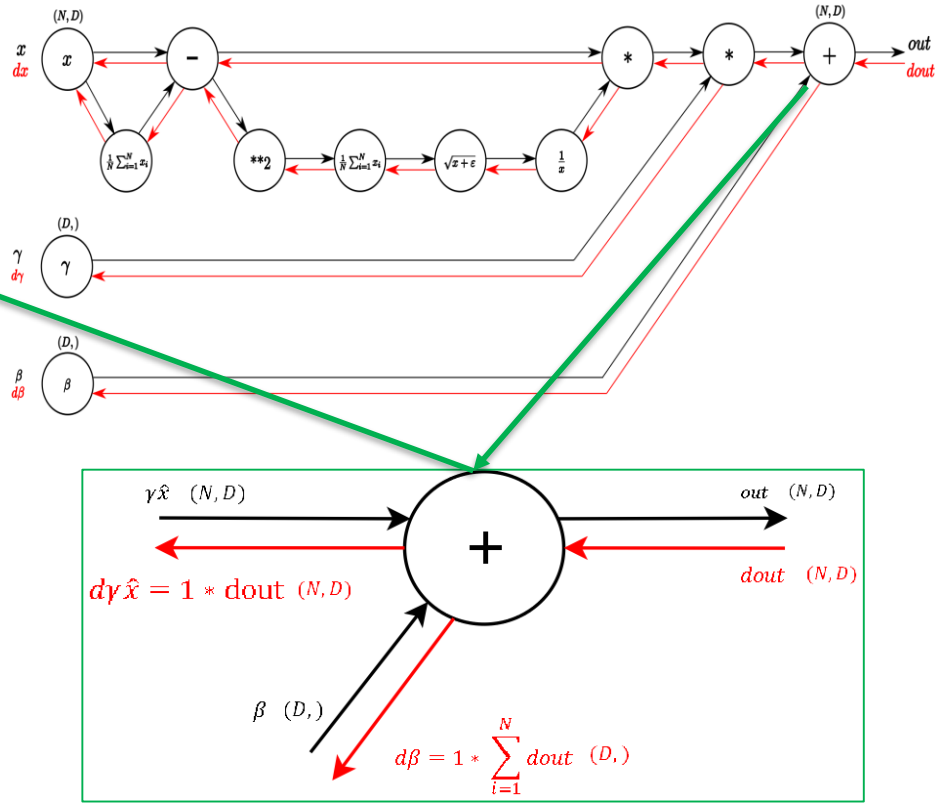
- Test Time: We use running mean and running variance instead of mean and variance of whole test data

```
x = (x - running_mean) / np.sqrt(running_var)
out = x * gamma + beta
```

Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

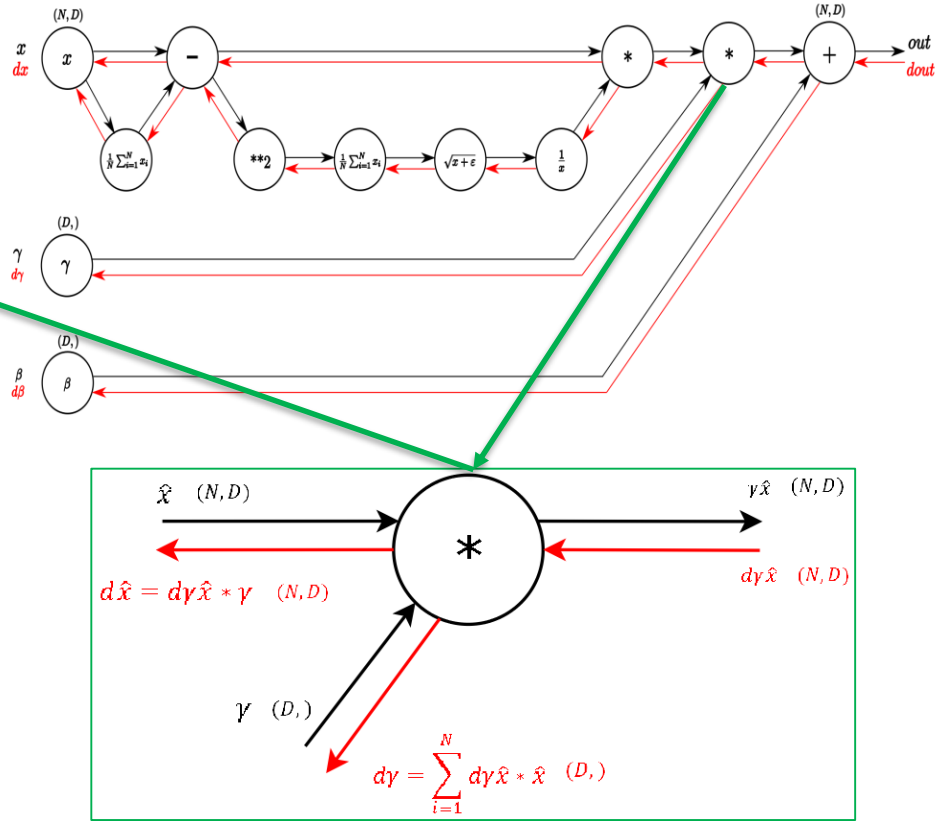
```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```




Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

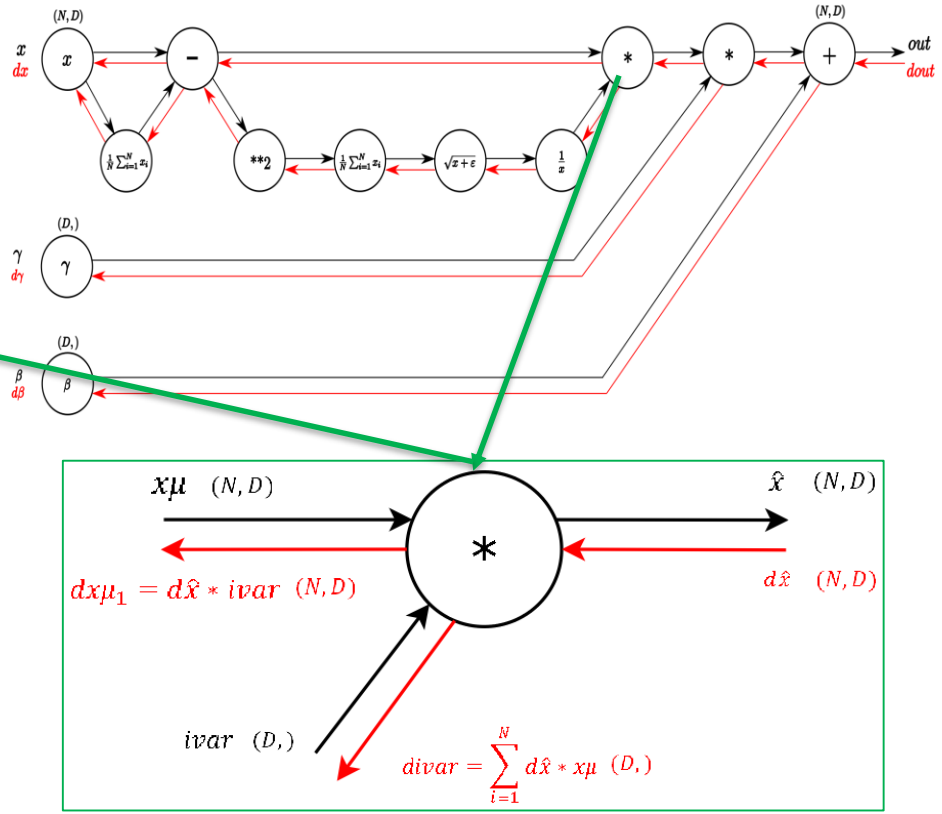
```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```




Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

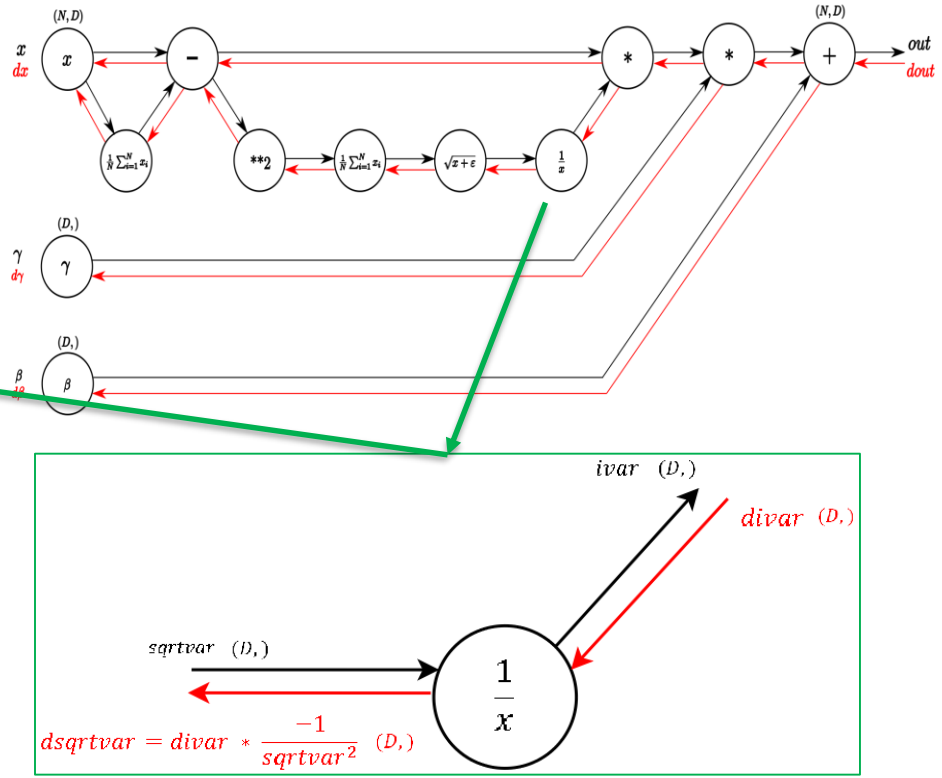
```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```




Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

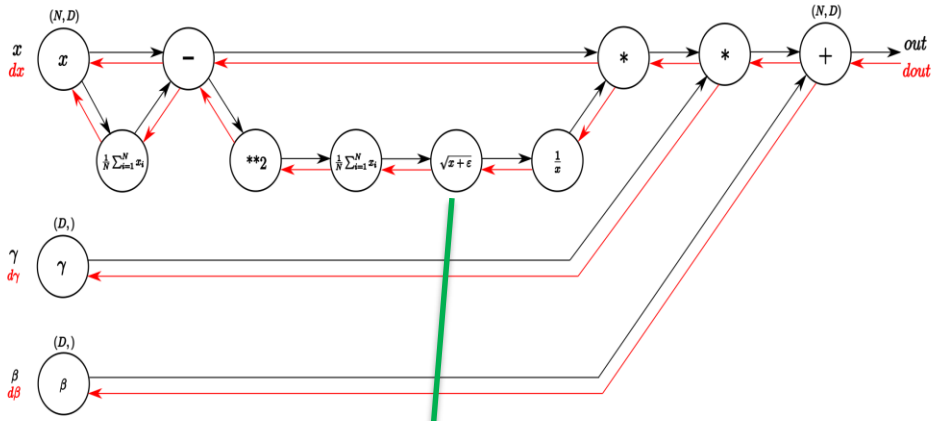
```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```

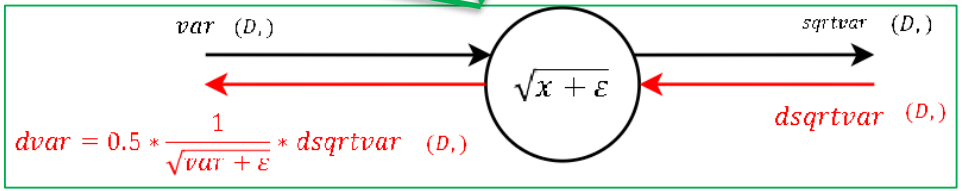



Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```

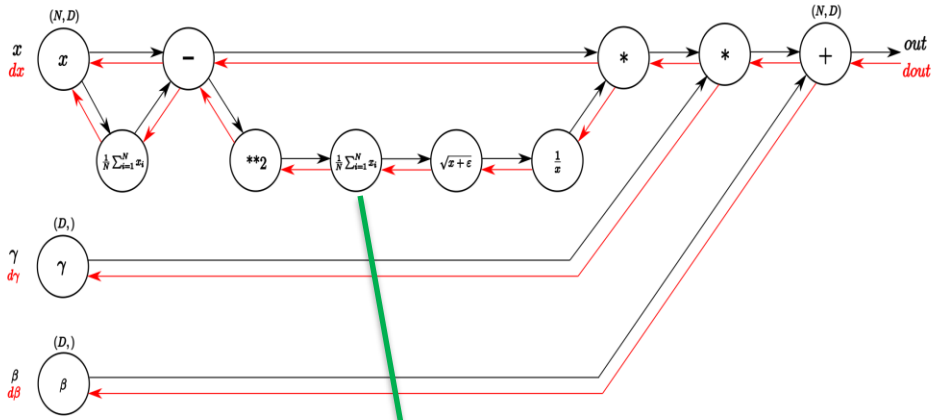


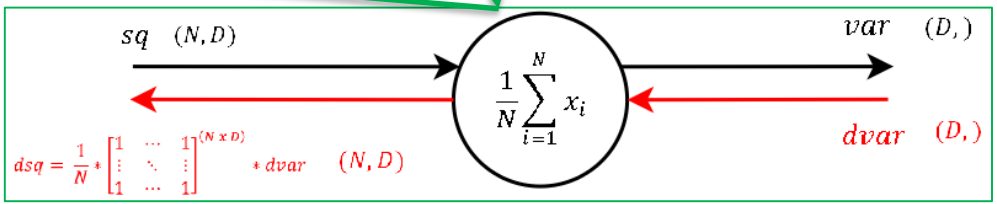




Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```

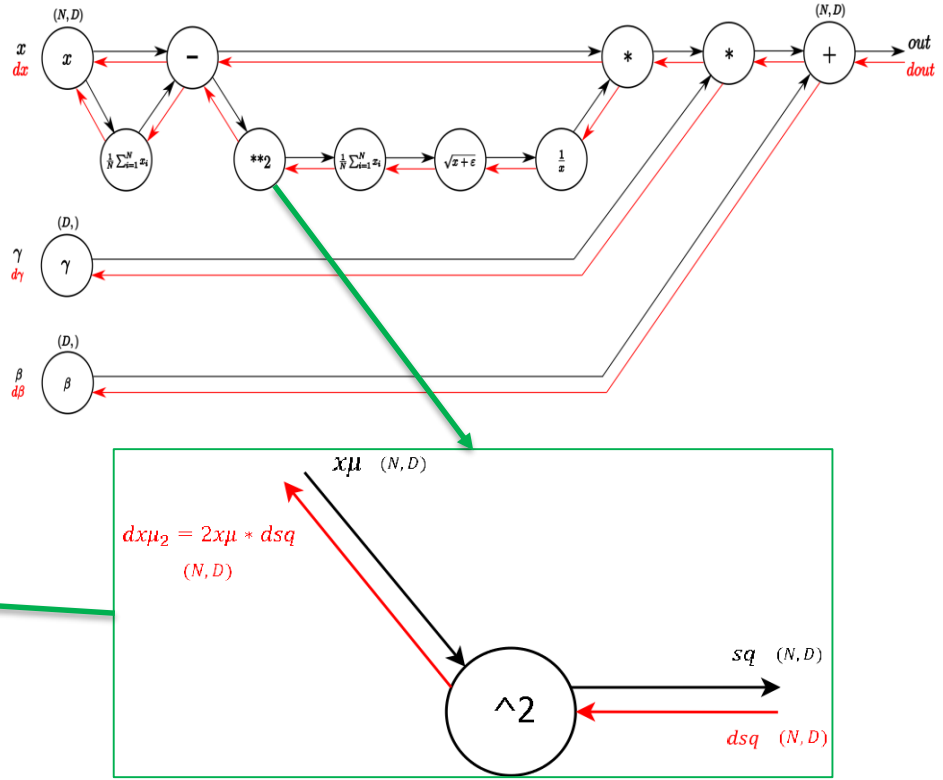



Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

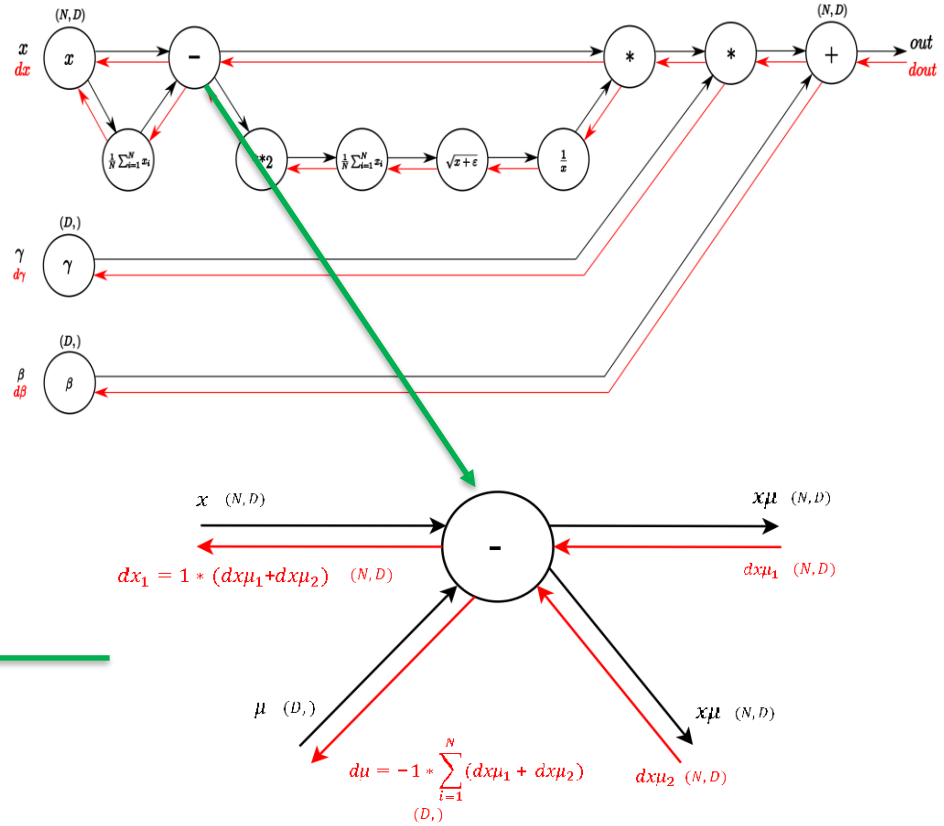
```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```




Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

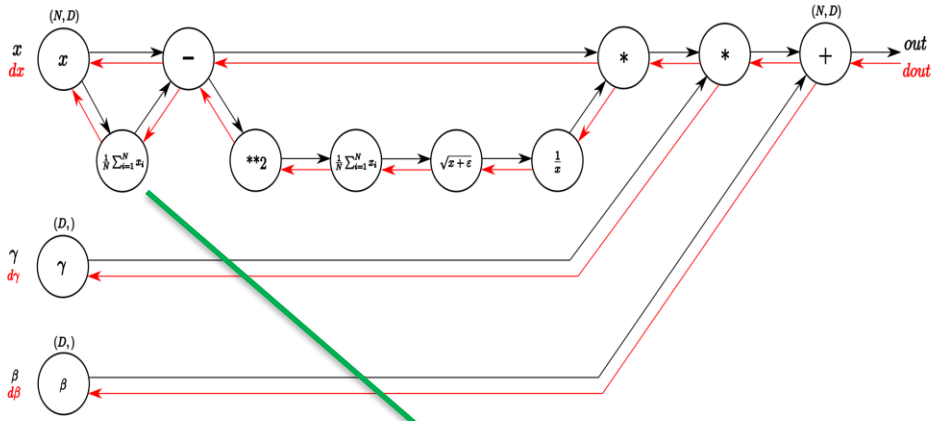
```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```

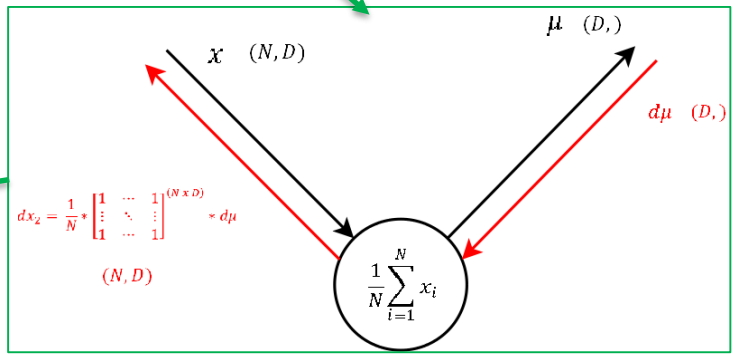



Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

# Batchnorm - Backward Pass

```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```

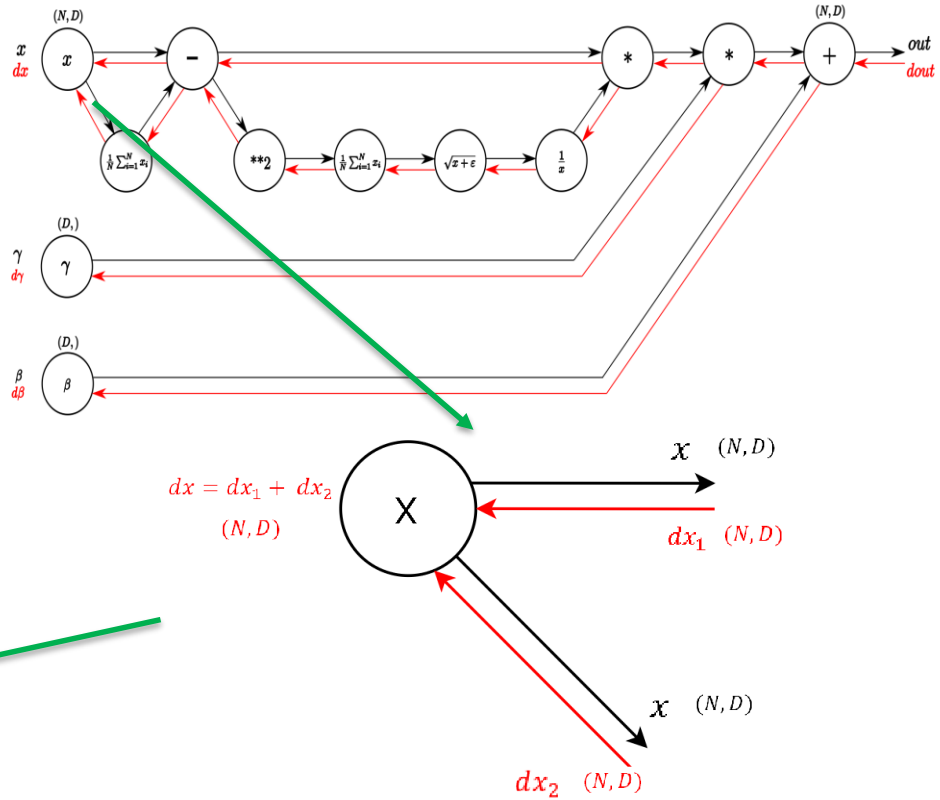



Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

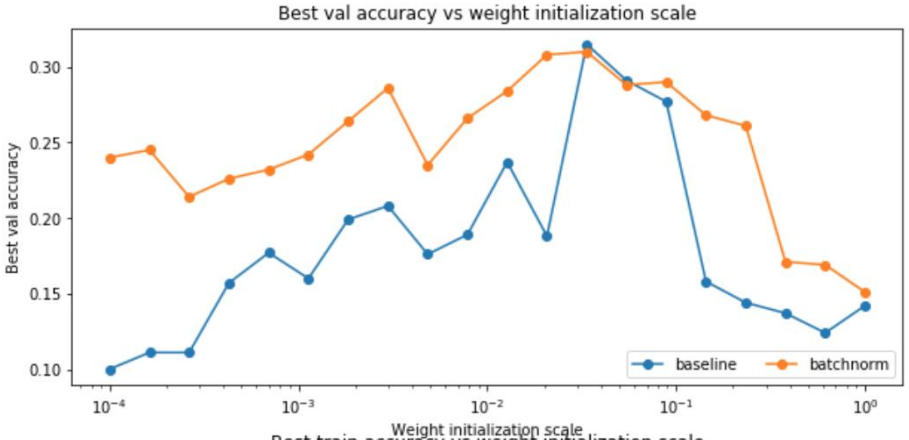
# Batchnorm - Backward Pass

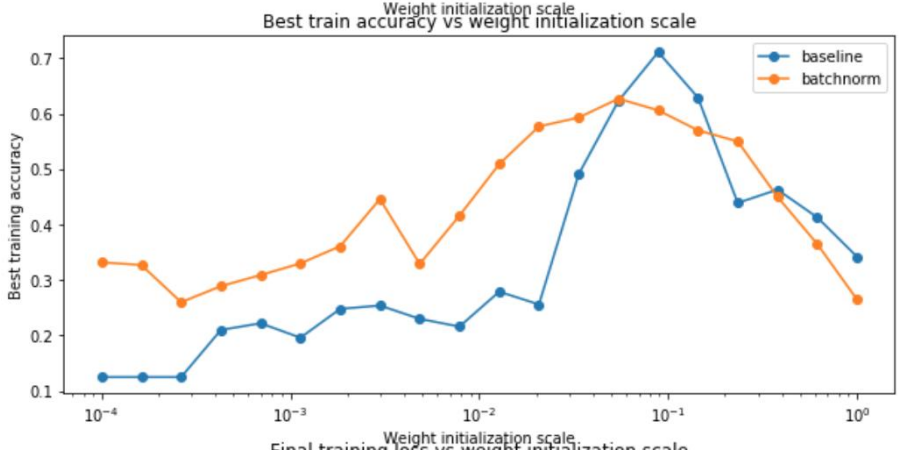
```
N, D = dout.shape
out, x_norm, beta, gamma, xmu, ivar, sqrtvar, var, eps = cache

# step9
dbeta = np.sum(dout, axis=0)

# step8
dxnorm = dout * gamma
dgamma = np.sum(dout * x_norm, axis=0)

# step7
divar = np.sum(dxnorm * xmu, axis=0)
dxmu1 = dxnorm * ivar

#step6
dsqrtvar = -1. / (sqrtvar ** 2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var + eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmean = -1. * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmean

#step0
dx = dx1 + dx2
```

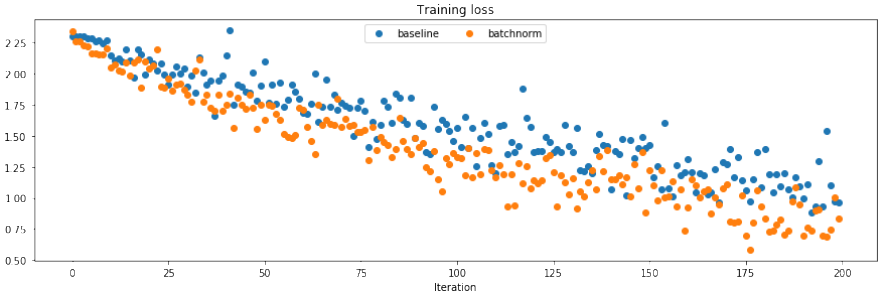


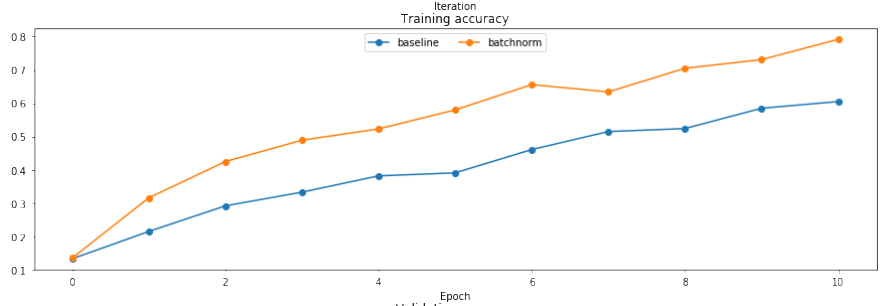

Source: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

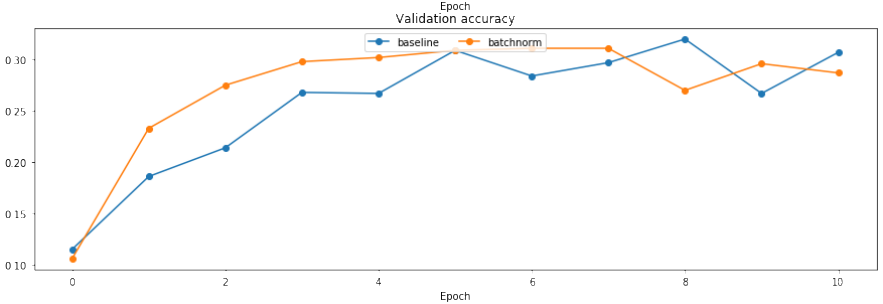
# BN - Results


Best val accuracy vs weight initialization scale
Best val accuracy
Weight initialization scale
baseline
batchnorm


Best train accuracy vs weight initialization scale
Best training accuracy
Weight initialization scale
baseline
batchnorm


Training loss
baseline
batchnorm
Iteration


Training accuracy
baseline
batchnorm
Epoch


Validation accuracy
baseline
batchnorm
Epoch

# Strategies for improvement?

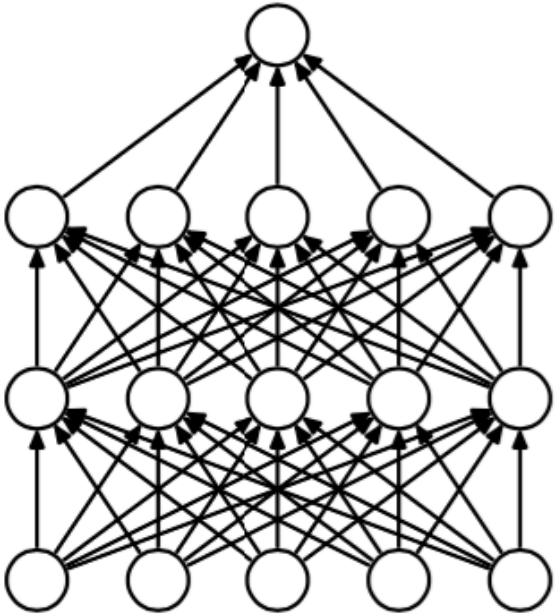
- More effective optimiser
  - SGD + Momentum
  - ADAM (https://arxiv.org/pdf/1412.6980.pdf)

- Batch Normalization

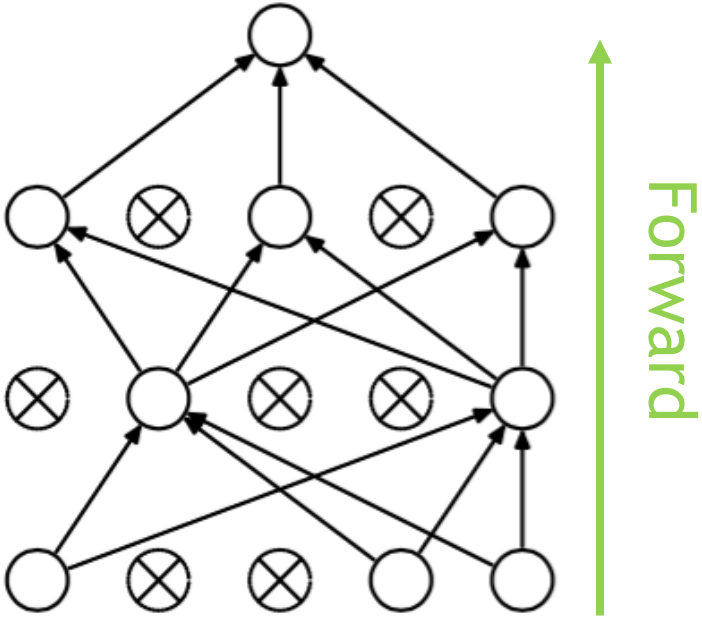
- Dropout

# Dropout

- Disable a random set of neurons (typically 50%)



(a) Standard Neural Net




(b) After applying dropout.

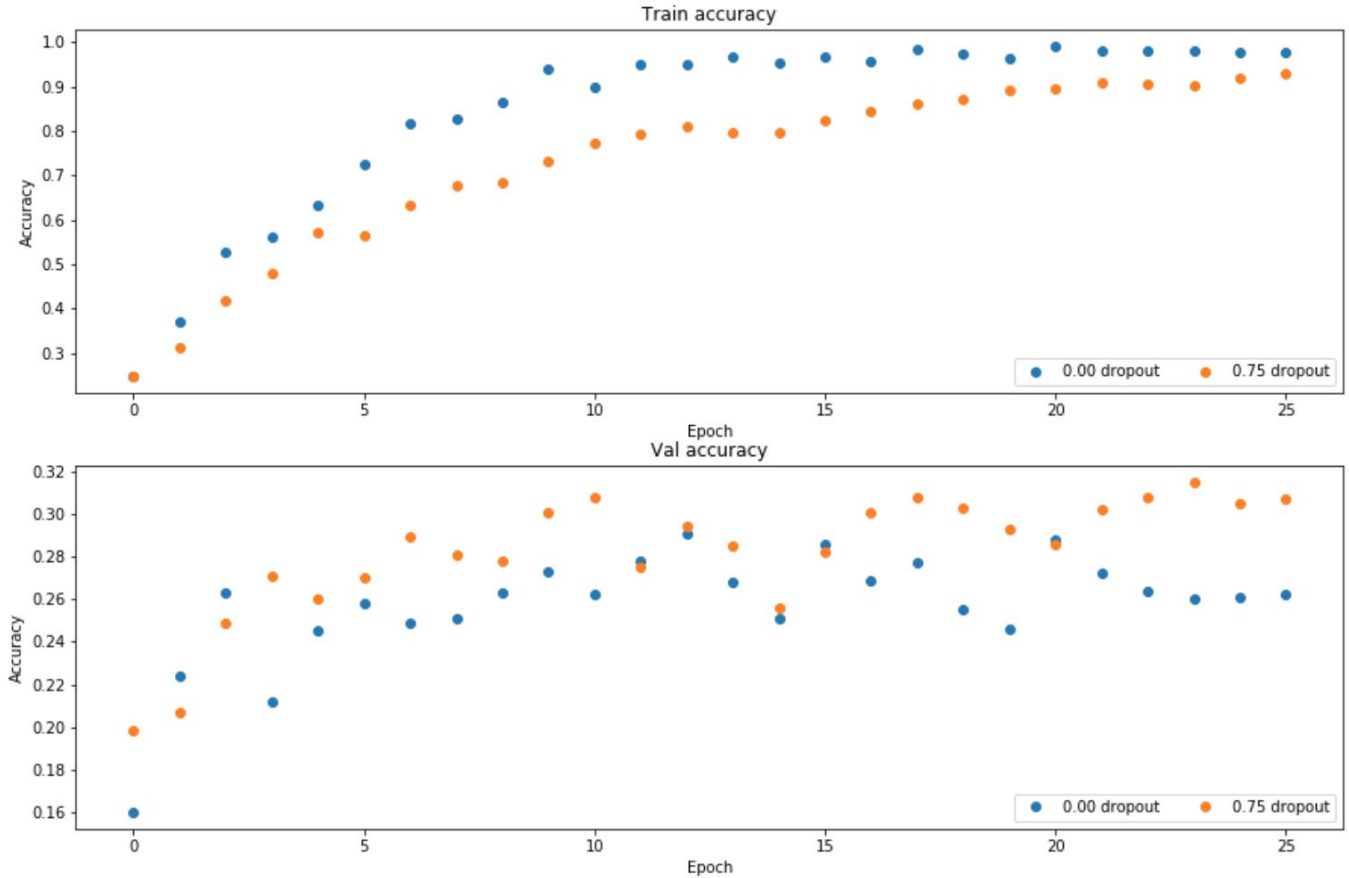
Srivastava 2014

# Dropout

```
if mode == 'train':
    ###########################################################################
    # TODO: Implement the training phase forward pass for inverted dropout.   #
    # Store the dropout mask in the mask variable.                            #
    ###########################################################################
    mask = (np.random.rand(*x.shape) > p) / (1 - p)
    out = x * mask

    ###########################################################################
    #                            END OF YOUR CODE                             #
    ###########################################################################
elif mode == 'test':
    ###########################################################################
    # TODO: Implement the test phase forward pass for inverted dropout.       #
    ###########################################################################
    out = x
    ###########################################################################
    #                            END OF YOUR CODE                             #
    ###########################################################################
```

Weight scaling inference rule

# Dropout


Train accuracy
Accuracy
Epoch
0.00 dropout
0.75 dropout
Val accuracy
Accuracy
Epoch
0.00 dropout
0.75 dropout

# Dropout – Why?

- Trainacc -> lower
- Valacc -> higher

-> Algorithm trains on a „different model“ everytime

-> Similar to network ensambles

# Strategies for improvement?

- More effective optimiser
  - SGD + Momentum
  - ADAM (https://arxiv.org/pdf/1412.6980.pdf)
- Batch Normalization
- Dropout



# Lookout

- Exercise 3 starts **Today: 19.12.2019**
  - Topics: Segmentation, Facial keypoint extraction, RNNs
  - **Deadline: 23.1.2020 18:00**
- Today:
  - Introduction to Pytorch (Important! )
  - CNN layers (Covered in Lecture 9)
  - CNN classifier (Covered in Lecture 10)

# Your Task For Next Exercise

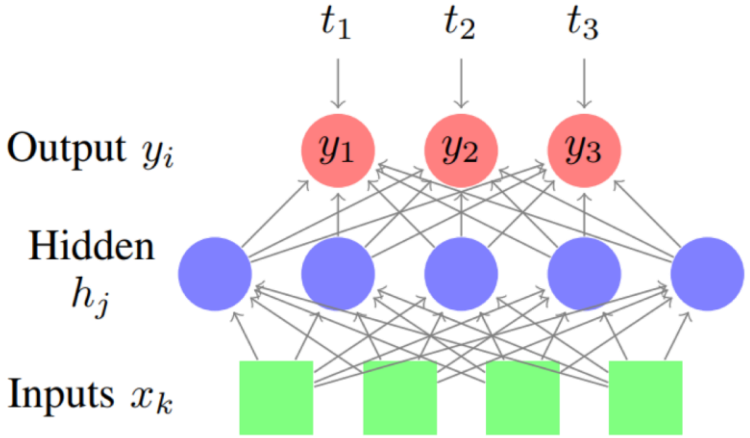
- Get accustomed to PyTorch!
- Try tutorials
- Don‘t be afraid to check out their implementations! What‘s different?

```
class Dropout(_DropoutNd):
    r"""During training, randomly zeroes some of the elements of the inp
    tensor with probability :attr:`p` using samples from a Bernoulli
    distribution. The elements to zero are randomized on every forward c

    This has proven to be an effective technique for regularization and
    preventing the co-adaptation of neurons as described in the paper
    `Improving neural networks by preventing co-adaptation of feature
    detectors`_ .

    Furthermore, the outputs are scaled by a factor of :math:`\frac{1}{1
    training. This means that during evaluation the module simply comput
    identity function.

    Args:
        p: probability of an element to be zeroed. Default: 0.5
        inplace: If set to ``True``, will do this operation in-place. De

    Shape:
```

# Questions?

# APPENDIX

# Derivatives of Cross Entropy Loss




Gradients of weights of last layer:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial p_i}\frac{\partial p_i}{\partial y_i}\frac{\partial y_i}{\partial w_{ji}} = -\sum t_i \frac{1}{p_i}\frac{\partial p_i}{\partial y_i}\frac{\partial y_i}{\partial w_{ji}}$$

$$E = -\sum t_i log\,(p_i) = -\sum t_i log\left(\frac{e^{y_i}}{\sum_k e^{y_k}}\right)$$

$$y_i = \sum_{j=1} h_j w_{ji}$$

# Derivatives of Softmax

$$
\begin{aligned}
\frac{\partial \frac{e^{y_i}}{\sum_k e^{y_k}}}{\partial y_j} &= \frac{e^{y_i} \sum_k e^{y_k} - e^{y_j} e^{y_i}}{\left(\sum_k e^{y_k}\right)^2} \\
&= \frac{e^{y_i} \left(\sum_k e^{y_k} - e^{y_j}\right)}{\left(\sum_k e^{y_k}\right)^2} \\
&= \frac{e^{y_j}}{\sum_k e^{y_k}} \times \frac{\left(\sum_k e^{y_k} - e^{y_j}\right)}{\sum_k e^{y_k}} \\
&= p_i(1 - p_j)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \frac{e^{y_i}}{\sum_k e^{y_k}}}{\partial y_j} &= \frac{0 - e^{y_j} e^{y_i}}{\left(\sum_k e^{y_k}\right)^2} \\
&= \frac{-e^{y_j}}{\sum_k e^{y_k}} \times \frac{e^{y_i}}{\sum_k e^{y_k}} \\
&= -p_j . p_i
\end{aligned}
$$

$$
\frac{\partial p_j}{\partial y_j} = \begin{cases} p_i(1 - p_j) & \text{if } i = j \\ -p_j . p_i & \text{if } i \neq j \end{cases}
$$

# Derivatives of Cross Entropy Loss

$$\frac{\partial E}{\partial p_i}\frac{\partial p_i}{\partial y_i} = -t_i(1-p_i) - \sum_{k\neq i} t_k \frac{1}{p_k}(-p_k.p_i)$$

$$= -t_i(1-p_i) + \sum_{k\neq i} t_k.p_i$$

$$= -t_i + t_i p_i + \sum_{k\neq i} t_k.p_i$$

$$= p_i\left(t_i + \sum_{k\neq i} t_k\right) - t_i$$

$$= p_i\left(t_i + \sum_{k\neq i} t_k\right) - t_i = p_i - t_i$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial p_i}\frac{\partial p_i}{\partial y_i}\frac{\partial y_i}{\partial w_{ji}}$$

$$= (p_i - t_i)\frac{\partial y_i}{\partial w_{ji}} = (p_i - t_i)h_j$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial p_i}\frac{\partial p_i}{\partial y_i}\frac{\partial y_i}{\partial w_{ji}} = -\sum t_i \frac{1}{p_i}\frac{\partial p_i}{\partial y_i}\frac{\partial y_i}{\partial w_{ji}}$$

$$\frac{\partial p_j}{\partial y_j} = \begin{cases} p_i(1-p_j) & \text{if } i = j \\ -p_j.p_i & \text{if } i \neq j \end{cases}$$