Deep Learning & Engineering Application

2. Backpropagation

JIDONG J. YANG
COLLEGE OF ENGINEERING
UNIVERSITY OF GEORGIA

Loss function

Per-sample loss

- $L(x, y, w) = C(y, f(x, w))$

Given a sample set:

- $S = \{(x_i, y_i), \quad i = 1, \dots, N\}$

The average loss over the sample:

- $L(S, w) = \frac{1}{N} \sum_{i=0}^{N-1} L(x_i, y_i, w)$

Gradient Descent

Full batch gradient (compute loss per the whole training sample set).

$$w \leftarrow w - \eta \frac{\partial L(S,w)}{\partial w}$$

Stochastic gradient descent (SGD)

- Pick a sample i, compute L, and update w.

$$w \leftarrow w - \eta \frac{\partial L(x_i, y_i, w)}{\partial w}$$

SGD exploits redundancy in the samples.

- Training goes faster, but very noisy
- In practice, use **mini-batches** for parallelization (GPU)

Momentum

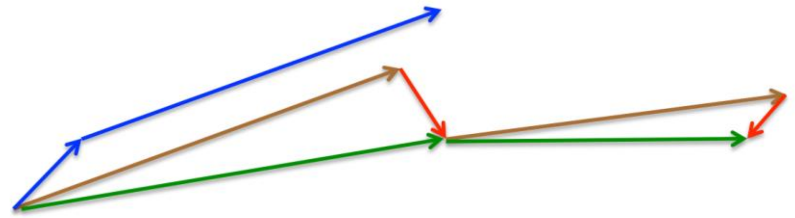
Regular momentum

- $v_{t+1} = \gamma v_t + \nabla f_i(w_t)$
- $w_{t+1} = w_t - \eta v_{t+1}$

Nesterov's momentum

- $v_{t+1} = \gamma v_t + \nabla f_i(w_t)$
- $w_{t+1} = w_t - \eta(\nabla f_i(w_t) + v_{t+1})$

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Demo: https://distill.pub/2017/momentum/

https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

References:
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

Backprop

Think about NN as a sequence of function compositions

$$Y = f_0(f_1(f_2 \ldots\ldots f_M(X)\ldots)$$

f can be:

- Linear, $\mathrm{Y = WX}$
- ReLU, $\mathrm{y = ReLU(x) = max[0, x]}$
- Duplicate (Distribute), $Y_1 = \mathrm{X}, \quad Y_2 = \mathrm{X}, \ldots\ldots, Y_k = \mathrm{X}$
- Add, $Y = X_1 + X_2$
- Maximum, $\mathrm{y} = \max(x_1, x_2)$
- LogSoftMax, $y_i = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right) = x_i - log\left[\sum_j \exp(x_j)\right]$

$$\frac{dC}{dX} = W^T \frac{dC}{dY} \qquad \frac{dC}{dW} = \frac{dC}{dY} X^T$$

$$\frac{dC}{dx} = \frac{dC}{dy}, x \geq 0 \qquad \frac{dC}{dx} = 0, x < 0$$

$$\frac{dC}{dX} = \frac{dC}{dY_1} + \frac{dC}{dY_2} + \cdots + \frac{dC}{dY_k}$$

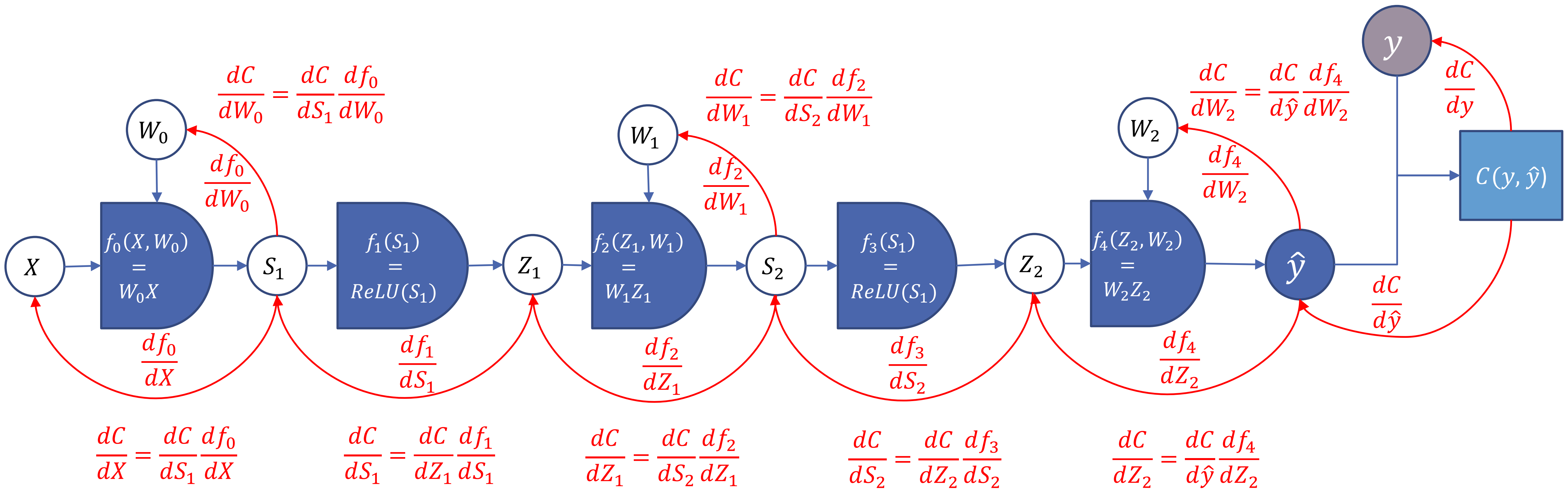
$$\frac{dC}{dX_1} = \frac{dC}{dX_2} = \frac{dC}{dY}$$

$$\frac{dC}{dx_1} = \frac{dC}{dy}, x_1 > x_2 \qquad \frac{dC}{dx_1} = 0, x_1 \leq x_2$$

- In Pytorch, nll_loss take log_softmax as input.
  e.g., output = F.nll_loss(F.log_softmax(input), target)
- Cross-entropy loss combines log_softmax and nll_loss in one single class and is implemented as:

```
        loss = nn.CrossEntropyLoss()
        output = loss(input, target)
or
        output = F.cross_entropy(input, target)
```

Backprop



$$\frac{dC}{dW_0} = \frac{dC}{dS_1}\frac{df_0}{dW_0}$$

$$\frac{dC}{dW_1} = \frac{dC}{dS_2}\frac{df_2}{dW_1}$$

$$\frac{dC}{dW_2} = \frac{dC}{d\hat{y}}\frac{df_4}{dW_2}$$

$$\frac{dC}{dX} = \frac{dC}{dS_1}\frac{df_0}{dX}$$

$$\frac{dC}{dS_1} = \frac{dC}{dZ_1}\frac{df_1}{dS_1}$$

$$\frac{dC}{dZ_1} = \frac{dC}{dS_2}\frac{df_2}{dZ_1}$$

$$\frac{dC}{dS_2} = \frac{dC}{dZ_2}\frac{df_3}{dS_2}$$

$$\frac{dC}{dZ_2} = \frac{dC}{d\hat{y}}\frac{df_4}{dZ_2}$$

Chain rule for vector functions

$Z_{in}: [d_{in} \times 1] \qquad Z_{out}: [d_{out} \times 1]$

$$\frac{\partial C}{\partial Z_{in}} = \frac{\partial C}{\partial Z_{out}} \frac{\partial Z_{out}}{\partial Z_{in}}$$

$$[1 \times d_{in}] = [1 \times d_{out}] * [d_{out} \times d_{in}]$$

Gradient = Gradient * Jacobian

Jacobian matrix

- $\left(\frac{\partial Z_{out}}{\partial Z_{in}}\right)_{ij} = \frac{(\partial Z_{out})_i}{(\partial Z_{in})_j}$ *(i.e., partial derivative of i^{th} output wrt. j^{th} input)*

If we have a cascade of functions(modules), we keep multiplying the Jacobian matrices of all the modules going backward and we get the gradients w.r.t all the internal variables.

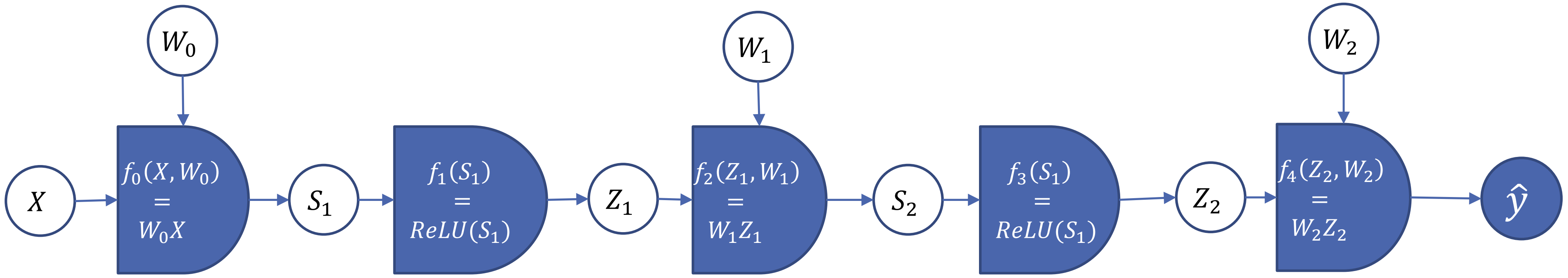
```
import torch
from torch import nn

image = torch.randn(2,28,28)
d0=image.nelement()
class mynet(nn.Module):
    def __init__(self, d0,d1,d2,d3):
        super().__init__()
        self.f0 = nn.Linear(d0,d1)
        self.f1 = nn.Linear(d1,d2)
        self.f2 = nn.Linear(d2,d3)
    def forward(self, x):
        z0 = x.view(-1)  #this flatten input
        s1 = self.f0(z0)
        z1 = torch.relu(s1)
        s2 = self.f1(z1)
        z2 = torch.relu(s2)
        y = self.f2(z2)
        return y
model = mynet(d0,60,40,10)
out = model(image)
out
```

PyTorch Implementation

Object-oriented

Use nn.Linear class (which includes a bias vector)

Use torch.relu function



Torch.autograd

Forward pass:

- Use the model’s prediction and the corresponding label to calculate the error (loss).

    prediction = model(data)

    loss = L(prediction, labels).sum()

Backward pass:

- Backpropagate this error through the network.
- Backward propagation is kicked off when we call .backward() on the error tensor.
- Autograd then calculates and stores the gradients for each model parameter in the parameter’s .grad attribute.

    loss.backward()

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

Problems with gradient descent

The greatest gradient doesn’t always point to the minimum.

Computational challenge with full batch

Noisy signal with stochastic gradient

Vanishing/exploding gradient

Adaptive Methods

A global learning rate may not work well as the gradients often vary between layers.

Use an estimate of a better rate separately for each weight.

Typically adapt to:

- Variance of the weights
- Local curvature

RMSprop and Adam

RMSprop - normalize by the root-mean-square of the gradient

- $v_{t+1} = \alpha v_t + (1-\alpha)\nabla f_i(w_t)^2 \longleftarrow$ Exponential moving average
- $w_{t+1} = w_t - \frac{\eta}{\sqrt{v_{t+1}+\epsilon}}\nabla f_i(w_t)$

Adam – adaptive moment estimation (RMSprop with a kind of momentum)

- $m_{t+1} = \beta m_t + (1-\beta)\nabla f_i(w_t) \longleftarrow$ Momentum
- $v_{t+1} = \alpha v_t + (1-\alpha)\nabla f_i(w_t)^2$
- $\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta^t}$
- $\hat{v}_{t+1} = \frac{v_{t+1}}{1-\alpha^t}$

Bias-correction

- $w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}+\epsilon}}\hat{m}_{t+1}$

RMSprop - Tieleman & Hinton [2012] https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
Adam - Kingma & Ba [2015] https://arxiv.org/pdf/1412.6980.pdf

Practical Aspects

Adam is often better than SGD, especially for poorly conditioned problems.

Adam is generally recommended over RMSprop due to the advantages of momentum.

Adam does not converge on some problems and produces worse generalization error on many computer vision problems.

Adam requires more memory than SGD.

In practice, try “SGD + Momentum” and “Adam” with different learning rates.

Gradient Clipping

Gradient Clipping-by-value

- $\hat{g} \leftarrow \frac{\partial L(S,w)}{\partial w}$
- $if\ \|\hat{g}\| \geq g_{ub}\ \ or\ \ \|\hat{g}\| \leq g_{lb}$
  - $\hat{g} \leftarrow g_{ub}\ \ or\ \ g_{lb}$

Gradient Clipping-by-norm

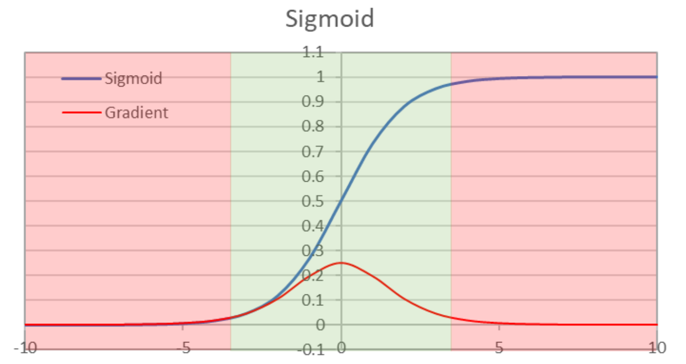
- $\hat{g} \leftarrow \frac{\partial L(S,w)}{\partial w}$
- $if\ \|\hat{g}\| \geq threshold$
  - $\hat{g} \leftarrow \frac{threshold}{\|\hat{g}\|}\hat{g}$

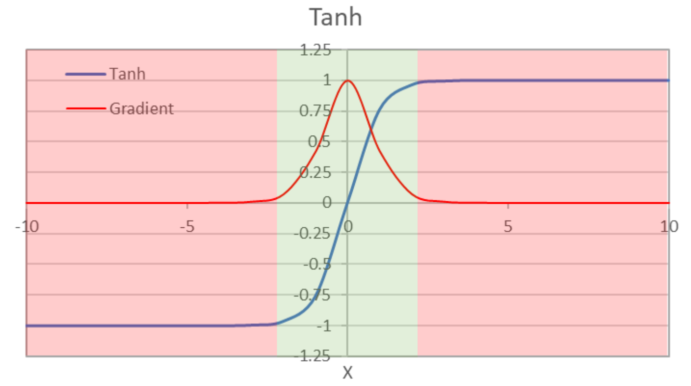
```
#PyTorch Implementation:

#Forward pass
predictions = model(X)
loss = criterion(predictions, labels)

Backward
optimizer.zero_grad()
loss.backward()

#Gradient clipping (by value)
nn.utils.clip_grad_value_(model.parameters(), clip_value=1.0)

#Gradient clipping (by value)
nn.utils.clip_grad_norm_(model.parameters(), max_norm=2.0, norm_type=2)

optimizer.step()
```

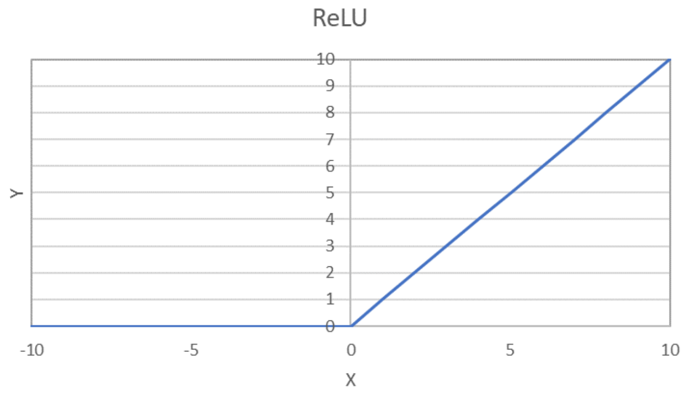
[Pascanu et al., 2013] https://arxiv.org/pdf/1211.5063.pdf

Activation Functions

- In traditional neural networks, sigmoid function is very popular activation function due to its differentiability and nice interpretation. Another popular one is tanh, which is a rescaled sigmoid.
- ReLU became dominant due to its computational advantage and nice properties, such as **better gradient propagation**, efficient computation, and scale invariant. (one of the most popular activation functions for deep neural networks)
- Variants of ReLU:
    - Parametric or Leaky ReLU, Exponential Linear Unit, etc.
- Why is the nonlinear activation function needed?
    - To approximate arbitrarily complex functions and learn complex features (both are naturally nonlinear).
    - Resemble the mechanism of brain neurons.
    - Without the non-linearity introduced by the activation function, multiple layers of a neural network are equivalent to a single layer neural network.







$$\mathbf{y}_l = \mathrm{W}_l\mathbf{x}_l + \mathbf{b}_l$$

Weight Initialization

Xavier initialization

Kaiming Initialization

- takes into account the non-linearity of ReLU.

A proper initialization method should avoid reducing or magnifying the magnitudes of input signals exponentially.

Forward Propagation

$$Var[y_l] = n_l Var[w_l x_l]$$

$$\Big\downarrow \quad w_{l-1}\text{: zero mean}$$

$$Var[y_l] = n_l Var[w_l] E[x_l^2]$$

$$\Big\downarrow \quad E[x_l^2] = \frac{1}{2} Var[y_{l-1}]$$

w_{l-1}: zero mean & symmetric
b_{l-1} = 0
activation = ReLU.

$$Var[y_l] = \frac{1}{2} n_l Var[w_l] Var[y_{l-1}]$$

$$\Big\downarrow$$

$$Var[y_L] = Var[y_1]\left(\prod_{l=2}^{L} \frac{1}{2} n_l Var[w_l]\right)$$

$$\frac{1}{2} n_l Var[w_l] = 1 \qquad Var[w_l] = \sqrt{2/n_l}$$

Backward Propagation

$$\frac{1}{2} \hat{n}_l Var[w_l] = 1 \qquad Var[w_l] = \sqrt{2/\hat{n}_l}$$

https://arxiv.org/pdf/1502.01852v1.pdf

https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Commonly-used Regularization Techniques

Penalty on weight parameters (e.g., L2 norm –> Weight Decay)

Weight sharing for convolutional neural networks (imposing some “structure”)

Dropout

etc.

Weight Decay

$$J_r = J(\boldsymbol{w}|X, y) + \alpha\Omega(\boldsymbol{w})$$

Let $\Omega(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$

$$J_r = J(\boldsymbol{w}|X, y) + \frac{\alpha}{2}\|\boldsymbol{w}\|_2^2$$

$$\nabla_{\boldsymbol{w}} J_r = \nabla_{\boldsymbol{w}} J + \alpha\boldsymbol{w}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\nabla_{\boldsymbol{w}} J_r = \boldsymbol{w} - \eta(\nabla_{\boldsymbol{w}} J + \alpha\boldsymbol{w}) = (1 - \eta\alpha)\boldsymbol{w} - \eta\nabla_{\boldsymbol{w}} J$$

Dropout

To prevent overfitting problems, dropout was proposed as an effective regularizer (Srivastava, et al., 2014).

Typically, dropout is conducted in a random fashion which is equivalent to sampling many "thinned" networks from the original network (sort of similar to parallel ensemble).

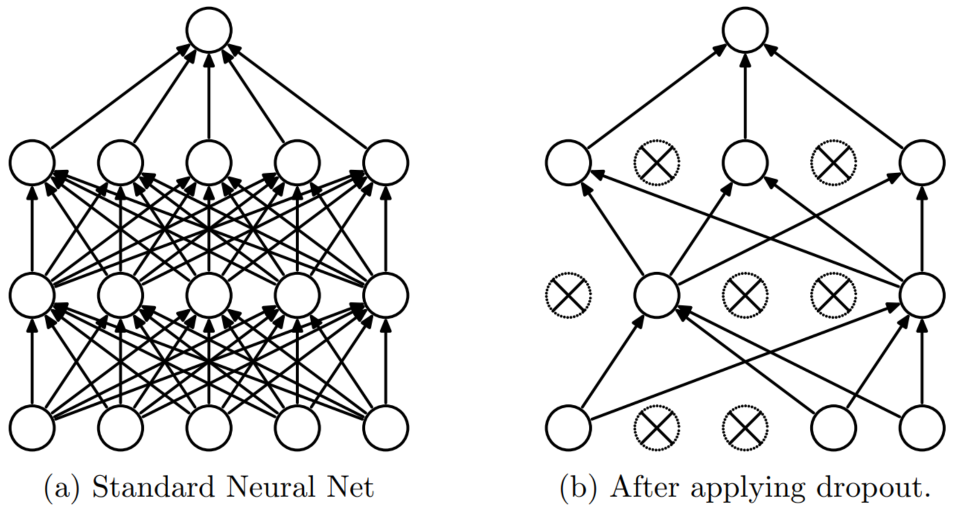


Figure 3-18 Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped. (Srivastava, et al., 2014)

Reference: Srivastava, N. et al., 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research,* 15(1), pp. 1929-1958

Dropout Neural Network Model

Without dropout

$$
\begin{aligned}
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)}\mathbf{y}^l + b_i^{(l+1)} \\
y_i^{(l+1)} &= f(z_i^{(l+1)})
\end{aligned}
$$

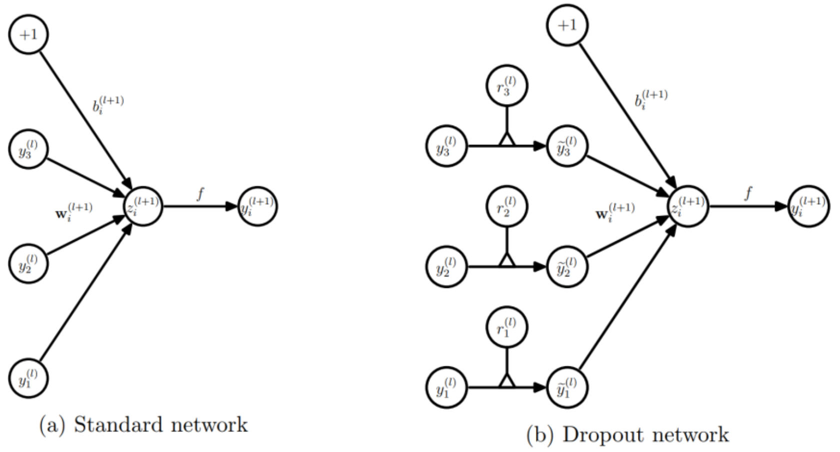


Figure 3: Comparison of the basic operations of a standard and dropout network.

With dropout

$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p) \\
\widetilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)} \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)}\widetilde{\mathbf{y}}^l + b_i^{(l+1)} \\
y_i^{(l+1)} &= f(z_i^{(l+1)})
\end{aligned}
$$

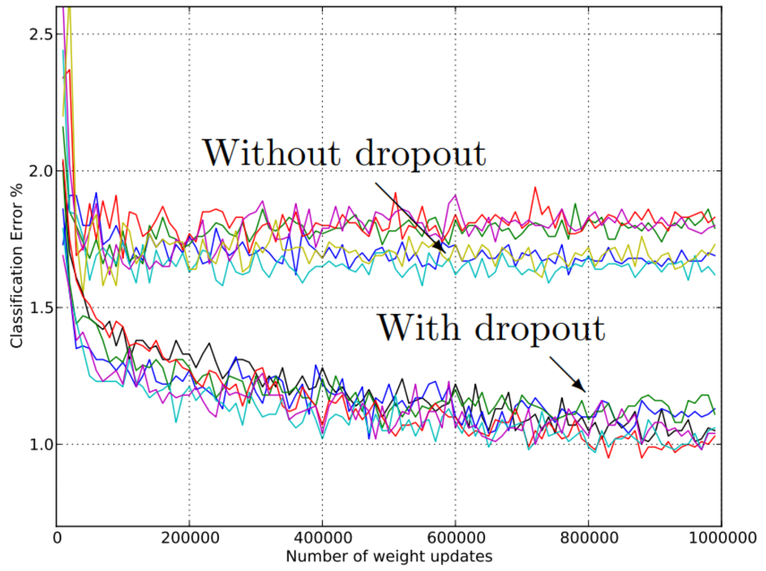


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Reference: Srivastava, N. et al., 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research,* 15(1), pp. 1929-1958

Backprop in Practice

Use ReLU non-linearities (instead of tanh and logistic)

Use cross-entropy loss for classification

Use Stochastic Gradient Descent on minibatches (e.g., SGD + Moment, Adam)

Shuffle the training samples

Normalize the input variables (zero mean, unit variance)

Schedule to decrease the learning rate

Gradient clipping

Proper weight initialization (e.g., Kaiming initialization)

Use regularization on the weights (magnitude and/or structure)

Use dropout

etc.