CENG502 – A Review of Deep Learning Fundamentals

Sinan Kalkan, Ozgur Aslan, Adnan Harun Dogan

March 2022

1 Problem Definition

Let us assume that we have a supervised learning setting and we are provided a set of N input-output pairs $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), ..., (\mathbf{x}_N, \mathbf{y}_N)\}$. In this supervised setting, we are interested in finding the function f that provides the mapping from the input space $\mathcal{X}$ to the output space $\mathcal{Y}$:

$$f : \mathcal{X} \rightarrow \mathcal{Y}. \tag{1}$$

In deep learning, f is a parametric function with parameters θ, and we are interested in finding θ:

$$\mathbf{y} = f(\mathbf{x}; \theta). \tag{2}$$

1.1 Empirical Risk Minimization

Finding θ in $\mathbf{y} = f(\mathbf{x}; \theta)$ is achieved by solving the following optimization (Empirical Risk Minimization) problem:

$$\theta^* \leftarrow \arg\min_\theta E\left[\mathcal{L}(f(\mathbf{x}; \theta), \mathbf{y})\right], \tag{3}$$

where $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ is a loss function measuring the dissimilarity of the prediction, $\hat{\mathbf{y}} = f(\mathbf{x}; \theta)$, with the correct output $\mathbf{y}$.

Alternatively, one could define and minimize other objectives, such as Structural Risk Minimization, Robust Risk Minimization, Invariant Risk Minimization etc. to find θ. Structural Risk Minimization, which includes a penalty term over the parameters, is especially relevant for deep learning:

$$\theta^* \leftarrow \arg\min_\theta E\left[\mathcal{L}(f(\mathbf{x}; \theta), \mathbf{y})\right] + \mathcal{R}(\theta), \tag{4}$$

where $\mathcal{R}(\theta)$ is e.g. $\sum_i \theta_i^2$.

2 Loss Functions

2.1 0-1 Loss

0-1 Loss measures the error of a deep model by counting the number of instances where there is an error:

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} \mathbb{1}(\mathbf{y}_{ij} \neq f(\mathbf{x}_i; \theta)_j), \tag{5}$$

which, however, is not differentiable and therefore, we use its approximations for regression and classification problems.

2.2 Loss Functions for Regression Problems

For a regression problem, we can use e.g. mean squared error ($\mathbf{y}_i$ has C dimensions):

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} (\mathbf{y}_{ij} - f(\mathbf{x}_i; \theta)_j)^2, \tag{6}$$

or mean absolute error:

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} |\mathbf{y}_{ij} - f(\mathbf{x}_i; \theta)_j|, \tag{7}$$

which are special and widely-used forms of:

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} |\mathbf{y}_{ij} - f(\mathbf{x}_i; \theta)_j|^q. \tag{8}$$

2.3 Loss Functions for Classification Problems

For a classification problem, we can e.g. use cross-entropy loss (for single-label classification problem):

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{i=1}^{N} -\log p(\mathbf{y}_{ij} | \mathbf{x}_i; \theta), \tag{9}$$

where $\mathbf{y}_{ij}$ is the correct class for $\mathbf{x}_i$, and $p(\mathbf{y}_{ij}|\mathbf{x}_i; \theta)$ can be easily obtained using the Softmax function:

$$p(\mathbf{y}_{ij} | \mathbf{x}_i; \theta) = \frac{\exp[f(\mathbf{x}_i; \theta)_j]}{\sum_k \exp[f(\mathbf{x}_i; \theta)_k]}. \tag{10}$$

Another frequently used loss function for classification problems is hinge loss (also called margin loss, max-margin loss):

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}\max(0, f_j - f_{y_i} + \Delta), \quad (11)$$

with $f_j = f(\mathbf{x}_i; \theta)_j$, and Δ is the to-be-satisfied margin between the score of the correct class y_i and the score of another class.

3 Gradient Descent and Other Optimizers

3.1 True, Stochastic, Batch Gradient Descent

True Gradient Descent (TGD) computes the gradient of the loss calculated over the whole data set and updates the parameters:

$$\begin{aligned} &\mathbf{g} = \nabla\left(\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(f(x_i;\theta), y_i) + \mathcal{R}(\theta)\right), \\ &\theta \leftarrow \theta - \eta \cdot \mathbf{g}, \end{aligned} \quad (12)$$

where $\mathcal{R}$ is the regularization term, e.g. squared L2 norm or L1 norm. The gradient indicates the direction in which the loss is increased. Therefore, to minimize the loss (the general case in deep learning), the negative of the gradient is used. The learning rate decides the "step size" in the negative gradient direction.

Since TGD uses the whole data set for one update of the parameters, the computation cost for one update is high (increases linearly with the number of data), and it can get stuck to a sub-optimal solution.

Stochastic Gradient Descent (SGD) computes the gradient for each data point:

$$\theta \leftarrow \theta - \eta\nabla\mathcal{L}(f(x_i;\theta), y_i). \quad (13)$$

This way, there are more frequent updates to parameters and the model can converge with less computation. However, updating the model this way, the computed gradients become noisy and the training becomes unstable. Therefore, with SGD, a small learning rate is used.

Batch Gradient Descent (BGD) provides a balance between TGD and SGD and computes the gradient over mini-batches of the data set to update the parameters (k denotes the iteration):

$$\theta \leftarrow \theta - \eta\nabla\frac{1}{|B_k|}\sum_{i\in B_k}\mathcal{L}(f(x_i;\theta), y_i), \quad (14)$$

which combines the strengths of the above methods with less computation for each update compared to the true gradient descent and more stable updates compared to the stochastic gradient descent.

3.2 Gradient Descent with Momentum

Gradients and weight updates can be made more robust by keeping a running average of the past gradients and the current gradient, i.e. by adding momentum:

$$\begin{aligned} &v_t = \beta v_{t-1} - \eta\mathbf{g_t}, \\ &\theta_{t+1} \leftarrow \theta_t + v_t. \end{aligned} \quad (15)$$

Momentum speeds up convergence and decreases the oscillations in the updates. The hyper-parameter $\beta \in [0, 1)$ adjusts the importance given to the previous gradients. With a value of 0, normal gradient descent is obtained. Due to addition of the momentum term βv_{t-1}, the updates are not in the direction of the steepest descent.

The Nesterov momentum corrects the direction of updates by computing the gradient after the momentum term is added to the parameters:

$$\begin{aligned} &\mathbf{g_t} = \nabla\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(f(x_i;\theta_{t-1} + \beta v_{t-1}), y_i), \\ &v_t = \beta v_{t-1} - \eta\mathbf{g_t}, \\ &\theta_{t+1} \leftarrow \theta_t + v_t. \end{aligned} \quad (16)$$

3.3 Adaptive Optimizers

In most cases, the partial derivative of each parameter has different magnitude. Therefore using the same learning rate for each parameter update can be sub-optimal. This problem can be solved using an adaptive learning rate method which uses the accumulation of past gradients to scale the learning rate. This way each parameter can have its own learning rate.

RMSProp [] computes the running average of squared gradients:

$$\mathbf{r} \leftarrow p\mathbf{r} + (1-p)\mathbf{g} \odot \mathbf{g}, \quad (17)$$

where $\mathbf{g}$ is the gradient, $\mathbf{r}$ is the accumulated gradient, p is a decay hyper-parameter similar to β in momentum (Eq. **??**) and $\odot$ is element-wise multiplication. Then the parameters are updated as:

$$\theta \leftarrow \theta - \frac{\eta}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}. \quad (18)$$

Adadelta [] does not need a global learning rate and parameter updates have the same units as the parameters:

$$\begin{aligned}
&\mathbf{r} \leftarrow p\mathbf{r} + (1-p)\mathbf{g} \odot \mathbf{g}, \\
&\Delta\theta = -\frac{\sqrt{\delta + \mathbf{d}}}{\sqrt{\delta + \mathbf{r}}}, \\
&\mathbf{d} \leftarrow p\mathbf{d} + (1-p)\Delta\theta \odot \Delta\theta, \\
&\theta \leftarrow \theta + \Delta\theta.
\end{aligned} \tag{19}$$

In Adadelta, different from the RMSProp, the decaying averages of parameter updates $\mathbf{d}$ is used in the nominator for unit correction.

Adam [] estimates the moments of the gradient and uses these estimates to compute adaptive learning rates for each parameter. The first moment update:

$$\mathbf{m_1} \leftarrow \beta_1 \mathbf{m_1} + (1-\beta_1)\mathbf{g}, \tag{20}$$

The second moment update:

$$\mathbf{m_2} \leftarrow \beta_2 \mathbf{m_2} + (1-\beta_2)\mathbf{g} \odot \mathbf{g}. \tag{21}$$

Adam combines the advantages of Momentum and RMSProp algorithm and uses them in combination. The first moment update is same as the Momentum update, and the second moment update is the same as RMSProp update. Due to zero initialization of the moments, they are biased towards zero especially at the initial updates. To overcome this issue, a correction step follows the update:

$$\hat{\mathbf{m}}_\mathbf{1} \leftarrow \frac{\mathbf{m_1}}{1-\beta_1^t}, \hat{\mathbf{m}}_\mathbf{2} \leftarrow \frac{\mathbf{m_2}}{1-\beta_2^t}, \tag{22}$$

where the decay parameters β_1, β_2 are raised to power t which is the time step parameter, increased after every update. The parameters are updated as:

$$\theta \leftarrow \theta - \eta \frac{\hat{\mathbf{m}}_\mathbf{1}}{\sqrt{\hat{\mathbf{m}}_\mathbf{2} + \delta}}. \tag{23}$$

In some tasks such as object detection, adaptive optimizers perform worse than SGD with momentum. This is because adaptive optimizers such as Adam fail to generalize as well as SGD with momentum. To overcome this issue, AdamW algorithm [31] decouple the weight decay that used as regularization (in most cases implemented as squared L2 norm) and gradient update term. They argue that the adaptive optimizers fail to better generalize due to the adaptive learning rate. Normally, with regularization, if a weight's value is high, the gradient update penalizes the weight to become smaller. However, in the same case, the adaptive optimizers will make the learning rate smaller, breaking the regularization effect. In most cases the gradient is computed as:

$$\mathbf{g} = \nabla(\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(f(x_i;\theta), y_i) + \|\theta\|_2^2).$$

AdamW [] decouples this computation:

$$\begin{aligned}
&\mathbf{g} = \nabla\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(f(x_i;\theta), y_i), \\
&\theta \leftarrow \theta - \eta(\frac{\hat{\mathbf{m}}_\mathbf{1}}{\sqrt{\hat{\mathbf{m}}_\mathbf{2} + \delta}} + \lambda\theta),
\end{aligned} \tag{24}$$

where $\hat{\mathbf{m}}_\mathbf{1}, \hat{\mathbf{m}}_\mathbf{2}$ are the same bias-corrected moments from Adam algorithm. With decoupling, the weight decay is not affected by the adaptive learning rates, resulting in better generalization.

4 Linear Classification and Regression

In linear classification and regression, our function $f(\cdot;\theta)$ is a simple linear function (for one output variable):

$$f(\mathbf{x}_i;\theta)_j = \sum_{k=1}^{D} w_{jk}\mathbf{x}_{ik} + b_j = \mathbf{w}_j \cdot \mathbf{x}_i + b_j, \tag{25}$$

where D is the dimensionality of $\mathbf{x}_\cdot$.

4.1 Interpretation

In a linear classifier or regressor, the function $f : \mathcal{X} \to \mathcal{Y}$ is actually a linear combination of its input. This implies that:

- The decision boundary between a class and other classes can only be a line (hyper-plane). If the classes are not linearly separable, a linear classifier will fall short in providing a good classification performance (under-fitting).
- As $\mathbf{w}_j \cdot \mathbf{x}_i$ is maximized when $\mathbf{w}_j \sim \mathbf{x}_i$, $\mathbf{w}_j$ should be a prototype (template) of the samples of class j to be able to provide highest scores for its samples.
- For a regression problem, a linear model implies that we are trying to fit a line (hyper-plane) through the data points in the training set. If the data is not distributed linearly, then the regression performance will be poor (under-fitting).

5 Non-linearity, Activation Functions

5.1 Squashing Activation Functions

Sigmoid is the oldest and was the most frequently used activation function:

$$\sigma(x) = \frac{1}{1+\exp(-x)}, \tag{26}$$

which has the benefit of having a simple derivative:

$$\sigma'(x) = \sigma(x)(1-\sigma(x)). \tag{27}$$

Hyperbolic tangent (tanh) is a version of sigmoid that's symmetrical around 0 vertically:

$$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)} = 2\sigma(2x)-1, \tag{28}$$

with a simple derivative like the sigmoid:

$$\tanh'(x) = 1-\tanh(x)^2. \tag{29}$$

Sigmoid and tanh have the benefit of squashing $(-\infty, \infty)$ into a small range. However, their values saturate very quickly as $|x| \to \infty$, which leads to a very small gradient to properly train a deep network (a.k.a. the vanishing gradient problem).

5.2 ReLU and its Variants

Rectified Linear Unit (ReLU) [33] is a simple thresholding operation on x:

$$\text{relu}(x) = \max(0, x), \tag{30}$$

which is fast to compute both during forward pass and backward pass. Its lack of saturation and vanishing gradient problems yields significant boosts in training deep networks, both in terms of training speed and generalization performance. Although its form may appear arbitrary, there are biological [15] and theoretical motivations (it is an approximation to the average of infinitely many sigmoids):

$$\text{relu}(x) \sim \log(1+\exp(x)) = \sum_{n=1}^{\infty} \sigma(x+0.5-n). \tag{31}$$

Despite its benefits, a ReLU unit may "die": For $x < 0$, it provides zero output during forward pass and makes the backward pass through that neuron effectively useless. Although this can be handled by keeping the learning rate small, its extensions have been proposed over the years. For example, Leaky ReLU [32]:

$$\text{relu}(x) = \max(\alpha, x), \tag{32}$$

which yields a non-zero small value, α, (a hyperparameter) if $x < 0$. Parametric ReLU [18] that treats α as a learnable parameter. Moreover, a whole linear model can be employed for both sides of the max function (a.k.a. maxout [17]):

$$\max(\mathbf{w}_1 \cdot \mathbf{x} + b_1, \mathbf{w}_2 \cdot \mathbf{x} + b_2), \tag{33}$$

which has the disadvantage of doubling the number of parameters.

5.3 Newer Activation Functions

Softplus:

$$\begin{aligned} \text{softplus}(x) &= ln(1+\exp(x)), & (34)\\ \text{softplus}'(x) &= \sigma(x). & (35) \end{aligned}$$

Swish [38]:

$$\text{swish}(x) = x\sigma(x). \tag{36}$$

Exponential linear unit [4]:

$$\text{elu}(x) = \begin{cases} x, & x \geq 0 \\ \alpha(\exp(x)-1), & x < 0 \end{cases} \tag{37}$$

6 Multi-layer Perceptrons (MLPs)

6.1 Forward Pass

In MLPs, different from the Linear Classification/Regression functions, multiple layers of linear functions followed by non-linearity functions are used to process the data. This way the MLPs can learn to approximate complex functions. For example a 2-layer MLP (1 hidden layer, 1 output layer):

$$\begin{aligned} h_{ij} &= \sigma(\mathbf{w}_j^h \cdot \mathbf{x}_i + b_j) = \sigma(net_{ij}^h), \\ \hat{y}_{ic} &= \sigma(\mathbf{w}_c^o \cdot \mathbf{h}_i + b_j) = \sigma(net_{ij}^o), \end{aligned} \tag{38}$$

where h_{ij} is the j-th neuron in the hidden layer for the i-th input, $\hat{y}_{ic}$ is the c-th output neuron for the i-th input and σ is the sigmoid non-linearity. In forward pass with the processing of the input some intermediate values that will be used in backward pass is stored.

6.2 Backward Pass: Backpropagation

To update the parameters of MLPs, the most efficient way of computing the gradients of each parameter is Backpropagation. In Backpropagation, gradients are computed layer by layer in the reverse order using the chain rule. For forward pass (Eq. 38) with one sample, Backpropagation is computed as:

$$\begin{aligned}
\mathcal{L}_i(\theta) &= \frac{1}{2}\sum_{c\in C}(\hat{y}_{ic} - y_{ic})^2, \\
\delta^o_{ic} &= \frac{\partial \mathcal{L}_i}{\partial net^o_{ic}} = \frac{\partial \mathcal{L}_i}{\partial y_{ic}}\frac{\partial y_{ic}}{\partial net^o_{ic}}, \\
&= (\hat{y}_{ic} - y_{ic})\hat{y}_{ic}(1-\hat{y}_{ic}), \\
\delta^o_{ij} &= \frac{\partial \mathcal{L}_i}{\partial net^h_{ij}} = \frac{\partial \mathcal{L}_i}{\partial h_{ij}}\frac{\partial h_{ij}}{\partial net^h_{ij}}, \\
&= \left(\sum_{c\in C}\frac{\partial \mathcal{L}_i}{\partial net^o_{ic}}\frac{\partial net^o_{ic}}{\partial h_{ij}}\right) h_{ij}(1-h_{ij}), \\
&= \left(\sum_{c\in C}\delta^o_{ic} w_{cj}\right) h_{ij}(1-h_{ij}).
\end{aligned} \tag{39}$$

Then the parameters are updated as:

$$\begin{aligned}
w^o_{ck} &= w^o_{ck} - \eta\delta^o_{ic}h_{ik}, \\
w^h_{jk} &= w^h_{ck} - \eta\delta^h_{ij}x_{ik},
\end{aligned}$$

where w^o_{ck} is the parameter connecting k-th hidden neuron to c-th output neuron and w^h_{jk} is the parameter connecting k-th input neuron to j-th hidden neuron.

6.3 Limitations of MLPs

Although MLPs have a vast capacity to approximate any function with some error [6, 19], they can be hard to train and it can be extremely difficult (if not impossible) to converge to practical solutions. The reasons are:

1. **Curse of dimensionality**: In machine learning, the number of samples required for obtaining small error increases exponentially with the dimensionality of the input. This is a significant limitation for the use of –especially shallow– MLPs on large inputs, e.g. high-resolution images [36].

2. **Equivariance and Invariance**: A function is equivariant if $f(g(x)) = g(f(x))$; i.e. applying function $f()$ on a transformation of x, i.e. $g(x)$, is equivalent to applying the transformation on $f()$ on original input, x. For example, an image segmentation network should be equivariant and for a spatially translated x by δ, the segmentation map should be translated by the same amount.

   An invariant function is, on the other hand, provides the same output even when the input is transformed: $f(g(x)) = f(x)$. An object recognition network should have such invariance: When we translate the image by δ pixels, the recognized objects should not change.

   MLPs, unfortunately, are difficult to train to obtain equivariance or invariance.

3. Images are vectorized when using MLPs, and vectorization breaks continuity of patterns, making it difficult and requiring more data to associate information belong to e.g. objects while training MLPs.

7 Overfitting, Memorization

7.1 Bias-Variance Dilemma

The bias-variance tradeoff is a fundamental concept in machine learning that describes the tension between the model's ability to fit the training data (low bias) and its ability to generalize to new, unseen data (low variance). A model with high bias will underfit the training data, while a model with high variance will overfit the training data. The goal is to find the right balance between bias and variance to achieve good generalization performance [13]. The bias-variance tradeoff can be formally defined as follows:

$$\epsilon = \mu^2 + \sigma + \bar{\epsilon}, \tag{40}$$

where μ is bias, σ is variance, ϵ is error, and $\bar{\epsilon}$ is irreducible error. The error is the expected squared difference between the predicted and true values, the bias is the expected difference between the model's predictions and the true values, the variance is the expected squared deviation of the model's predictions from their expected value, and the irreducible error is the inherent noise in the data. To reduce bias, we can use a more complex model or increase the number of training samples. To reduce variance, we can use a simpler model or increase the regularization.

7.2 Regularization

Regularization is a common technique used to prevent overfitting in deep learning models. It involves adding a penalty term to the loss function that encourages the model weights to be small. Two popular regularization techniques are L1 and L2 regularization. L1 regularization adds a penalty term proportional to the absolute value of the weights, while L2 regularization adds a penalty term proportional to the square of the weights. The L1 and L2 regularization can be expressed mathematically as:

$$L_1 := L_{\text{data}} + \lambda \sum_i |w_i|, \tag{41}$$

$$L_2 := L_{\text{data}} + \frac{\lambda}{2} \sum_i w_i^2, \tag{42}$$

where λ is a hyperparameter that controls the strength of the regularization. L1 regularization is useful when we want to promote sparse solutions, while L2 regularization is useful when we want to reduce the magnitude of the weights [34].

7.3 Dropout and its Variants

Dropout is a regularization technique that helps to prevent overfitting in deep neural networks by randomly dropping out (i.e., setting to zero) some of the neurons during training. This forces the network to learn more robust and diverse features and prevents co-adaptation of neurons to each other. The dropout probability is typically set between 0.2 and 0.5 and is applied to each neuron independently with a different random mask for each training sample. Dropout has been shown to be effective in improving the generalization performance of deep neural networks and is widely used in many applications [43, 47].

Several variants of dropout have been proposed to enhance its performance further. For example, the DropConnect method extends dropout by randomly dropping out connections between neurons instead of entire neurons, which can be seen as a more general form of dropout. Another variant is the ShakeDrop regularization, which applies random transformations to the dropout masks and has been shown to improve the regularization effect. In addition, the Gaussian Dropout method applies Gaussian noise to the activations of the neurons instead of setting them to zero, which can provide smoother regularization and better training stability. Other variants of dropout include SpatialDropout, which applies dropout to entire feature maps in convolutional neural networks, and DropBlock, which drops out contiguous regions of activations to improve the robustness of the network. These variants have shown promising results in various tasks and can be used to complement the standard dropout regularization technique [49, 1].

7.4 Data Augmentation

Data augmentation is a popular technique used in deep learning to increase the size of a training dataset by artificially generating new data from existing data. This is done to help prevent overfitting and improve the generalization of a model. Common data augmentation techniques include flipping, rotation, scaling, and shifting of the images. Other techniques such as color jittering, adding noise, and cutout have also been used. These techniques can be applied to different types of data including images, audio, and text. For example, in image classification tasks, flipping and rotation can be used to generate new images from the original ones by randomly flipping and rotating them. The new images can be added to the training dataset, increasing its size and diversity [39].

Mathematically, data augmentation can be represented as a function f that maps an input sample x to an augmented sample x'. The function f can be defined using different transformations T applied to the input x. For example, for image data, T could be a combination of flipping, rotating, and scaling operations. The function f can then be defined as f(x) = T(x) + e, where e is a noise term. The noise term is added to the augmented sample to introduce some randomness and variability to the generated data [29].

Data augmentation has been shown to be an effective technique for improving the performance of deep learning models, particularly in situations where the amount of training data is limited. However, care should be taken when applying data augmentation, as some transformations may not be appropriate for certain types of data. For example, flipping or rotating a medical image may lead to incorrect diagnoses. Therefore, domain-specific knowledge should be taken into account when selecting the appropriate data augmentation techniques [5].

8 Misc Details

8.1 Preprocessing

Generally, subtraction from mean ($\mu_j = \sum_k \mathbf{x}_{kj}$) to zero-center the data:

$$\mathbf{x}_{ij} \leftarrow \mathbf{x}_{ij} - \mu_j, \tag{43}$$

and normalization with standard deviation ($\sigma_j = \sqrt{\sum_k (\mathbf{x}_{kj} - \mu_j)^2 / N}$) are helpful:

$$\mathbf{x}_{ij} \leftarrow \frac{\mathbf{x}_{ij} - \mu_j}{\sigma_j}. \tag{44}$$

8.2 Initialization

Initializing the weights is critical as, otherwise, variances of the activations and gradients across layers change, which affect the training performance [14]:

$$Var\left[\sum_{i=1}^{n} w_i x_i\right] = nVar[w]Var[x]. \tag{45}$$

Sampling $w_i \sim U\left[\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$ (to get rid of n in the variance) is not sufficient since this would lead to $nVar[w] = 1/3$, a constant scaling factor across layers (because variance of $U[-r, r]$ is $r^2/3$). To compensate that, we can have either of the following (a.k.a. Xavier initialization [14]):

$$w_i \sim U\left[\frac{-\sqrt{3}}{\sqrt{n}}, \frac{\sqrt{3}}{\sqrt{n}}\right], \tag{46}$$

$$w_i \sim N(0, 1/\sqrt{n}). \tag{47}$$

which need to be slightly adjusted if the number of inputs and outputs for a layer are different [14].

8.3 Normalization

In batch normalization [23], the activation of a neuron a is zero-centered and normalized over a batch B of m examples:

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} a_i, \tag{48}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(a_i - \mu_B)^2, \tag{49}$$

$$a_i \leftarrow \frac{a_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \tag{50}$$

$$a_i \leftarrow \gamma a_i + \beta, \tag{51}$$

where γ and β are learnable scaling and shifting parameters. Alternatively, normalization can be performed over all channels for a single input (layer norm), a single channel of a single instance (instance norm) or across a subset of channels (group norm).

8.4 Learning Rate

The learning rate determines the step size that the optimizer takes at each iteration to update the model parameters. Choosing an appropriate learning rate is crucial because if the learning rate is too small, the optimization process may converge too slowly, while if it is too large, the optimization process may fail to converge or even diverge. The learning rate can be set manually or adaptively during the training process. Some popular adaptive learning rate methods include AdaGrad, Adam, RMSProp, and Adadelta.

One approach to choosing an appropriate learning rate is to use a learning rate schedule, which decreases the learning rate over time. A common schedule is to decrease the learning rate by a factor of 10 after a fixed number of epochs or when the validation loss stops improving. Another approach is to use a learning rate range test, which involves gradually increasing the learning rate until the loss starts to increase sharply. The optimal learning rate is then chosen as the midpoint of the range where the loss is still decreasing. The cyclical learning rate policy is another technique that involves cyclically increasing and decreasing the learning rate to explore different regions of the optimization landscape[41, 42, 40].

There are several strategies for learning rate selection in deep learning:

- **Manual Selection:** Manually setting the learning rate is a common strategy. However, it is a time-consuming and tedious task, as the learning rate has a significant impact on the training process.
- **Learning Rate Schedules:** In this strategy, the learning rate is gradually decreased during the training process. Some common learning rate schedules are step decay, exponential decay, and polynomial decay. The learning rate schedule can be based on the number of epochs or the validation loss.
- **Adaptive Learning Rate Methods:** These methods adapt the learning rate during training, based on the performance of the model. Some popular adaptive learning rate methods are AdaGrad, Adam, and RMSprop.
- **Learning Rate Range Test:** This strategy involves running the model on a small range of learning rates to identify the optimal range for the learning rate. The learning rate is then set to the middle of this range.
- **Cyclic Learning Rates:** In this strategy, the learning rate is cycled between a minimum and maximum value during training. This can help the model converge faster and achieve better results.

These strategies can help in selecting an appropriate learning rate for the model and prevent issues like slow convergence or overfitting.

9 Evaluation and Model Selection

9.1 Evaluation measures

When evaluating the performance of a deep learning model, various measures can be used to assess its accuracy and effectiveness. These include metrics such as accuracy, precision, recall, F1 score, and ROC curve analysis. Each metric provides a different perspective on the model's performance and can be useful in different scenarios.

Accuracy is the proportion of correctly classified instances out of all instances and is given by:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{52}$$

Precision measures the proportion of true positive predictions out of all positive predictions and is given by:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{53}$$

Recall measures the proportion of true positive predictions out of all actual positive instances and is given by:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{54}$$

F1 score is the harmonic mean of precision and recall and is given by:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{55}$$

ROC curve analysis plots the true positive rate (TPR) against the false positive rate (FPR) for different classification thresholds and is given by:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad \text{and} \quad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{56}$$

9.2 Model Selection

Model selection involves selecting the best architecture and hyperparameters for a specific deep learning task. There are various approaches to model selection, each with its advantages and disadvantages. Some of the most commonly used methods include manual search, grid search, random search, and Bayesian optimization.

- **Manual search** involves manually selecting different model architectures and hyperparameters and evaluating them on a validation set. One simple example of a hyperparameter is the learning rate, which controls the size of the step taken during gradient descent. Another example is the number of hidden layers in a neural network. The researcher starts with a simple model and then gradually adds complexity until the desired level of performance is achieved.

- **Grid search** involves systematically searching a pre-defined set of hyperparameters to find the optimal combination. The pre-defined set is usually specified as a range of values for each hyperparameter. For example, the range of learning rates might be [0.001, 0.01, 0.1]. The grid search algorithm then evaluates the performance of each combination of hyperparameters and selects the one that performs the best.

- **Random search** involves randomly sampling hyperparameters from a pre-defined hyperparameter space. For example, the learning rate might be sampled from a uniform distribution over the range [0.001, 0.1]. The random search algorithm then evaluates the performance of each combination of hyperparameters and selects the one that performs the best.

- **Bayesian search** is a more advanced approach that uses a probabilistic model to predict the performance of different hyperparameters. Bayesian optimization starts with a prior belief about the performance of different hyperparameters and then updates the belief based on the observed performance of each combination of hyperparameters. The algorithm then selects the hyperparameters that are expected to perform the best based on the updated belief.

In conclusion, model selection is an essential step in deep learning, and choosing the right approach can significantly impact the performance of the model. Different methods have different strengths and weaknesses, and the choice of method should depend on the specific problem being addressed and the available resources. It is essential to carefully evaluate and compare different models to ensure the best possible performance.

Hold-out method, k-fold cross-validation, leave one subject out.

10 Convolutional Neural Networks (CNNs)

10.1 Motivation for CNNs

The solution to the problems discussed in Section 6.3 takes inspiration from the findings reported by

Hubel and Wiesel [22]: Neurons in visual cortex are not fully-connected and have local connectivity in a local receptive field. The first neural architecture to exploit this is the Neocognitron model [12]. The Neocognitron model was later extended to be trained with gradient descent and weight sharing to be applied to to more challenging problems [28], shaping the foundations of CNNs used nowadays.

CNNs can be trained with significantly less samples, addressing the curse of dimensionality problem [10]. They are equivariant to translation, though they do not provide any invariance to scaling or rotation.

10.2 Layers in CNNs

CNNs are constructed by combining the following operations:

- **Convolution**: Applies linear transformation on a window of the input and repeats the operation by sliding the window:

$$a_i^{(l+1)} = \sum_{j=1}^{F} w_j \cdot a_{i+j-1}^{(l)}, \tag{57}$$

  which illustrated convolution for a 1D layer with window size F. $a_{i+j-1}^{(l)}$ can have multiple dimensions, which are called channels, and in this case, the window needs to have dimensionality $F \times C$, where C is the number of channels.

- **Pooling**: Pooling layers are used to summarize activations and reduce dimensionality in a CNN. Although many alternatives exist, the simple max-pooling strategy works very well for a variety of problems:

$$a_i^{(l+1)} = \max(a_i^{(l)}, ..., a_{i+F-1}^{(l)}). \tag{58}$$

- **Non-linearity**: The non-linear functions discussed in Section 5 are used in CNNs as well. ReLUs are very common.

- **Normalization**: The normalization operations discussed in Section 8.3 are very helpful in CNNs.

- **Dense layers**: The last layers of CNNs are generally fully-connected layers mapping the activations of the last convolutional layers to the outputs desired.

10.3 Misc Architectural Choices

Residual networks, inception networks, deformable convolution, group convolution, separable convolution, ...

- **Residual connections:** Residual connections [3, 20] allow for the addition of the input to the output of one or more layers in a neural network. This approach has been shown to improve convergence rates, particularly for very deep networks. One of the main advantages of residual connections is that they enable the network to learn the identity mapping, which helps prevent the degradation of performance that can occur as the network depth increases. However, residual connections can also introduce additional computational overhead and may require careful tuning to achieve optimal performance.

- **Inception networks:** Inception networks [46, 45] use multiple parallel branches of convolutional filters with different receptive fields to capture features at multiple scales. This approach has been shown to be effective in image classification tasks, particularly when the input images contain objects of different sizes. One of the main advantages of Inception networks is that they enable the network to capture a wide range of features, from low-level edges and textures to high-level semantic information. However, Inception networks can be computationally expensive, particularly when used with large input images or very deep architectures.

- **Unshared convolution:** Unshared convolution [16, 26, 53, 44] applies each filter to a specific region of the input, rather than sliding the filter over the entire input. This approach has been shown to be useful in tasks where the spatial relationships between input features are important, such as object detection or semantic segmentation. One of the main advantages of unshared convolution is that it allows the network to capture fine-grained spatial information without introducing excessive computational overhead. However, unshared convolution can also result in a larger number of parameters, which can make the network more difficult to train and require more computational resources.

- **Atrous (dilated) convolution:** [27, 51, 52, 35] Atrous convolution increases the receptive field of each filter without adding extra parameters by inserting zeros in between the filter weights. This approach has been shown to be effective in image segmentation tasks, where a larger receptive field can help capture more contextual information. One of the main advantages of atrous convolution is that it can be used to increase the field of view without increasing the number of parameters or computational cost. However, using larger dilation rates can lead to a loss of spatial resolution, which can negatively affect performance.

- **Transposed convolution:** [11, 54, 37] Transposed convolution, also known as fractionally-strided convolution or deconvolution, allows for the upsampling of feature maps. This approach has been used in tasks such as image generation and semantic segmentation. One of the main advantages of transposed convolution is that it enables the network to produce high-resolution output from low-resolution input. However, transposed convolution can be computationally expensive and can introduce artifacts such as checkerboard patterns.

- **3D convolution:** 3D convolution extends the concept of convolution to volumetric data, such as video or medical imaging. This approach has been shown to be effective in tasks such as action recognition or brain image segmentation. One of the main advantages of 3D convolution is that it allows the network to capture spatial and temporal information simultaneously. However, 3D convolution can be computationally expensive and requires larger amounts of memory than 2D convolution.

- **1x1 convolution:** [48, 24, 2] 1x1 convolution applies a single filter to each pixel or feature vector independently of its neighbors. This approach has been shown to be useful in reducing the number of parameters and computational cost, particularly in deep architectures. One of the main advantages of 1x1 convolution is that it can be used to perform feature dimensionality reduction or channel-wise mixing, which can help capture more complex relationships between features. However, 1x1 convolution can also result in information loss, particularly when used excessively.

- **Separable convolution:** [30, 46, 44] Separable convolution is a type of convolution that breaks down the standard convolution operation into two simpler operations, namely depth-wise convolution and point-wise convolution. The depth-wise convolution is performed on each channel of the input tensor independently, while the point-wise convolution is applied to mix the output channels. By doing so, separable convolution significantly reduces the number of parameters and computations required compared to standard convolution, leading to faster and more efficient models. However, separable convolution may not always yield the best performance, especially for models with limited depth or in scenarios where accuracy is prioritized over speed.

- **Group convolution:** [25, 50, 21] Group convolution is a technique that partitions the input and output channels of a convolutional layer into multiple groups and performs convolution separately for each group. This allows for parallel processing of different parts of the input feature maps and enables more efficient computation on parallel hardware architectures. Additionally, group convolution can improve model generalization by promoting diversity among feature maps and reducing overfitting. However, choosing an appropriate group size can be challenging and may require extensive experimentation.

- **Deformable convolution:** [7, 56, 8] Deformable convolution is a variant of standard convolution that allows for dynamic sampling of the input feature maps based on learned offsets. By incorporating offset information, deformable convolution can better capture spatial structures and adapt to various object shapes and deformations. This makes it particularly useful for tasks such as object detection and segmentation. However, deformable convolution requires more computations and parameters than standard convolution, making it more computationally expensive and prone to overfitting.

- **Position-sensitive convolution:** [7, 9, 55] Position-sensitive convolution is a type of convolutional layer that utilizes position-sensitive score maps to modulate the feature maps before and after the convolution operation. By doing so, position-sensitive convolution can better capture object structure and improve localization accuracy, making it suitable for object detection tasks. However, position-sensitive convolution can be computationally expensive and may require large amounts of training data to learn effective score maps.

11 Recurrent Neural Networks (RNNs)

There are problems that require working with sequential (e.g. text or videos) data. Sequential data require summarizing and representing the history of data, $x_{t-T}, ..., x_{t-1}$, and making predictions based on this history and the current input x_t.

11.1 Recurrence and unfolding

Sequential data can be addressed by a network that captures the history (e.g. for a classification problem):

$$h_t = H(h_{t-1}, x_t) = \tanh(W^{xh} x_t + W^{hh} h_{t-1}) \quad (59)$$
$$y_t = Y(h_t) = \text{softmax}(W^{hy} h_t), \quad (60)$$

which form the basis of RNNs. The ability to story history, i.e. having memory, enables RNNs to be Turing complete; in other words, any problem that can be solved with a Turing machine can be represented by an RNN.

In an RNN, we have recurrence in calculating h_t, which is not suitable for vanilla feed-forward and back-ward pass in a deep network for learning with gradient descent. This can be alleviated by unfolding the recurrence relation for a fixed number of steps, which converts the recurrent architecture into a deep feed-forward network with weight-sharing across different time steps.

11.2 Exploding or Vanishing Gradient Problem

Solution 1: Gradient clipping and penalty on gradient norm to prevent vanishing gradients.

Solution 2: Use an explicit memory mechanism on which gradients don't vanish.

11.3 RNN Variants

Long Short Term Memory (LSTM)

$$a = W^x x_t + W^h h_t + b, \quad (61)$$
$$a_i, a_f, a_o, a_g \leftarrow a, \quad (62)$$
$$i = \sigma(a_i), f = \sigma(a_f), \quad (63)$$
$$o = \sigma(a_o), g = \tanh(a_g), \quad (64)$$
$$c_t = f \odot c_{t-1} + i \odot g, \quad (65)$$
$$h_t = o \odot \tanh(c_t). \quad (66)$$

Gated Recurrent Units (GRUs)

Machine translation as an example?

References

[1] Jimmy Ba and Brendan Frey. Adaptive dropout for training deep neural networks. *Advances in Neural Information Processing Systems*, 26:3084–3092, 2013.

[2] João Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[3] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[4] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[5] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. *arXiv preprint arXiv:1909.13719*, 2020.

[6] George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989.

[7] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. *Proceedings of the IEEE international conference on computer vision*, pages 764–773, 2017.

[8] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks v2: More deformable, better results. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9308–9317, 2020.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Ppm: Pyramid pooling module. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2881–2890, 2017.

[10] Simon S Du, Yining Wang, Xiyu Zhai, Sivaraman Balakrishnan, Ruslan Salakhutdinov, and Aarti Singh. How many samples are needed to learn a convolutional neural network? *arXiv preprint arXiv:1805.07883*, 2018.

[11] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[12] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[13] Stuart Geman, Elie Bienenstock, and René Doursat. Neural Networks and the Bias/Variance Dilemma. *Neural Computation*, 4(1):1–58, 01 1992.

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[15] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[17] Ian Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. In *International conference on machine learning*, pages 1319–1327. PMLR, 2013.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[19] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[20] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, ..., and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[21] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[22] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574, 1959.

[23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[24] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2013.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[27] Yann LeCun, Bernhard Boser, John S Denker, Donald Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. In *Neural computation*, volume 1, pages 541–551. MIT Press, 1989.

[28] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[29] Justin Lemley, Siavash Bazrafkan, and Jason J Corso. Smart augmentation learning an optimal data augmentation strategy. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1497–1505. IEEE, 2017.

[30] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[31] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[32] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Citeseer, 2013.

[33] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[34] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 78, New York, NY, USA, 2004. Association for Computing Machinery.

[35] Xuecheng Nie, Shuang Wang, Hongsheng Li, Xiaoyu Yang, Si Liu, Yi-Zhe Song, Timothy M Hospedales, and Chen Change Loy. Mutual learning to adapt for joint human parsing and pose estimation. *arXiv preprint arXiv:1803.09127*, 2018.

[36] T Poggio and Q Liao. Theory i: Deep networks and the curse of dimensionality. *Bulletin of the Polish Academy of Sciences. Technical Sciences*, 66(6), 2018.

[37] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2016.

[38] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[39] Christian Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[40] Leslie N Smith. Cyclical learning rates for training neural networks. In *IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[41] Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[42] Leslie N Smith and Nicolas Topin. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[43] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[44] Vivienne Sze, Yu-Hsin Chen, Tianjun Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[45] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2016.

[46] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[47] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. *CVPR*, 2015.

[48] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 4489–4497, 2015.

[49] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013.

[50] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[51] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2016.

[52] Fisher Yu and Vladlen Koltun. Dilated residual networks. *arXiv preprint arXiv:1705.09914*, 2017.

[53] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[54] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2528–2535, 2010.

[55] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaoyong Wang, and Jiaya Jia. Pyramid scene parsing network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2881–2890, 2017.

[56] Xiaoxiao Zhu, Han Hu, and Stephen Lin. Deformable convolutional networks: a review. *arXiv preprint arXiv:1906.08469*, 2019.